An Error-Recovering Form of DCGs*

JUKKA PAAKKI

Nokia Research Center
P.O.Box 156, 02101 Espoo, Finland

KARI TOPPOLA

Department of Computer Science, University of Helsinki
Teollisuuskatu 23, 00510 Helsinki, Finland

Abstract

In this paper an alternative implementation of Prolog's Definite Clause Grammars (DCGs) is presented. The DCG variant is based on the context-free grammar class LL(1) and it solves some of the problems with parsing programming languages using conventional DCGs, such as nondeterminism and intolerance to syntax errors.

1. DCGs and Context-Free Grammars

The programming language Prolog has been connected to parsing right from its very birth: the first real implementation of the logic programming idea [Col 73] was actually developed for processing (i.e. parsing) *natural languages*. Since then, several special notations especially for parsing have been introduced in Prolog, the most popular one being the Definite Clause Grammars (DCGs) [PeW80]. DCGs can be considered as an executable form of context-free grammars that have traditionally been the leading notation in specifying the syntax of *programming languages*.

Informally, a context-free grammar consists of a finite set of *nonterminal symbols*, a finite set of *terminal symbols*, and a finite set of *productions* of the form

$$A \longrightarrow S_1, S_2, \ldots, S_n (n >= 0)$$

* Lecture presented at the 1st Finnish-Hungarian Workshop on Programming Languages and Software Tools, Szeged, Hungary, August 8—11, 1989.

where A is a nonterminal symbol, and each S_i is either a nonterminal or a terminal symbol. A context-free grammar represents all the syntactically legal sentences (programs) of the language. A sentence can be derived from the grammar by beginning with a symbol string consisting of the designated *start symbol* and by repeatedly replacing a nonterminal in the symbol string with the right-hand side of a production for that nonterminal, until the string contains only terminal symbols; that terminal string is a sentence of the language. The language defined by the context-free grammar consists of exactly those sentences that can be derived from the start symbol.

As an example, simple arithmetic expressions can be defined with the following context-free grammar where the set of nonterminal symbols is {expr, term, factor, number}, the set of terminal symbols is {"+", "*", "(", ")", "0", "1", "2", "3", "4", "5", "6", "7", "8", "9"}, and the start symbol is expr:

```
expr   ----> expr, "+", term
expr   ----> term
term   ----> term, "*", factor
term   ----> factor
factor ----> "(",expr,")"
factor ----> number
number ----> "0"
number ----> "1"
number ----> "2"
number ----> "3"
number ----> "4"
number ----> "5"
number ----> "6"
number ----> "7"
number ----> "8"
number ----> "9"
```

DCGs, as a notation, resemble much context-free grammars. In a DCG, nonterminal symbols are represented by Prolog terms and terminal symbols by Prolog lists. For example, the context-free grammar given above can be modified into Quintus Prolog [Qui 86] simply by terminating each production with a period.

2. DCGs and Language Processing

The DCG facility is in most Prolog dialects implemented with a transformation from DCG into ordinary Prolog. The transformation is straightforward: each nonterminal is translated into a predicate with two extra arguments (representing the input symbol list before and after processing the corresponding nonterminal), and each terminal is translated into a call for a special built-in predicate (corresponding to advancing the input pointer to the next input symbol).

As an example, the DCG for simple arithmetic expressions outlined in chapter 1 would be translated into the following Prolog program (for clarity, we present the terminals explicitly, instead of using their ASCII codes):

```
expr(S0, S)    :- expr(S0, S1),shift(S1, '+', S2),term(S2, S).
expr(S0, S)    :- term(S0, S).
term(S0, S)    :- term(S0, S1), shift(S1, '*', S2), factor(S2, S).
term(S0, S)    :- factor(S0, S).
factor(S0, S)  :- shift(S0, '(', S1), expr(S1, S2), shift(S2, ')', S).
factor(S0, S)  :- number(S0, S).
number(S0, S):- shift(S0, '0', S).
number(S0, S):- shift(S0, '1', S).
number(S0, S):- shift(S0, '2', S).
number(S0, S):- shift(S0, '3', S).
number(S0, S):- shift(S0, '4', S).
number(S0, S):- shift(S0, '5', S).
number(S0, S):- shift(S0, '6', S).
number(S0, S):- shift(S0, '7', S).
number(S0, S):- shift(S0, '8', S).
number(S0, S):- shift(S0, '9', S).
```

Here shift is the built-in scanning predicate:

$$\text{shift}([X|S], X, S).$$

It can be interpreted as "removing symbol X from input stream $[X|S]$, producing stream S".

Sentences of a language are recognized by a parsing process. Most parsing strategies lay some restrictions on the underlying context-free grammar of the language: for instance ambiguous grammars are usually forbidden. Conventionally a DCG is applied, i.e. the input program is "parsed", by executing the corresponding ordinary Prolog program. The operational semantics of Prolog thus implies that a DCG implemented this way produces a top-down, left-to-right, recursive descent, backtracking parser. This characterization in terms of normal Prolog brings out some problems with DCGs when considering practical parsing of programming languages:

(1) the order of alternative productions for a nonterminal has great significance on the speed of the parser (parsing is nondeterministic),
(2) left-recursive grammars cannot be handled,
(3) no recognition or recovery of syntax errors is provided, and
(4) lexical analysis cannot be interleaved with parsing (since the source program is represented as a list of symbols); this leads to two passes over the source program for parsing it.

On the other hand, reducing DGCs into ordinary Prolog makes them more general than context-free grammars:

(i) grammar symbols can have an arbitrary number of arguments, and
(ii) procedure calls can be embedded within productions.

These additional features make DCGs closely related with attribute grammars [Knu 68]: arguments can be considered as "attributes" and procedure calls as "semantic rules".

As an example, our DCG for arithmetic expressions can be revised in such a way that the value of an expression is evaluated during parsing. Note that the original

version is left-recursive; we have to remove left-recursion and for instance replace it with right-recursion in order to make the DCG correctly executable. The arguments represent the values of the subexpressions, and procedure calls are enclosed in {...}.

```
expr(Val)    ---> term(V1), "+", expr(V2), {Val is V1+V2}.
expr(Val)    ---> term(Val).
term(Val)    ---> factor(V1), "*", term(V2), {Val is V1*V2}.
term(Val)    ---> factor(Val).
factor(Val)  ---> "(", expr(Val),")".
factor(Val)  ---> number(Val).
number(0)    ---> "0".
number(1)    ---> "1".
number(2)    ---> "2".
number(3)    ---> "3".
number(4)    ---> "4".
number(5)    ---> "5".
number(6)    ---> "6".
number(7)    ---> "7".
number(8)    ---> "8".
number(9)    ---> "9".
```

3. A More Practical Form of DCGs

We have implemented the DCG formalism in a way that is more related to the parsing theory of context-free grammars. Most notably, we have tried to remove the shortcomings (1)—(3) of the conventional DCG implementation strategy discussed in the previous chapter. The initial idea was to support primarily syntax error handling, but the resulting system was expected to contribute to other parsing aspects as well, such as efficiency. In the sequel we shall briefly present the main characteristics of the system.

Determinism

Since the normal execution model in Prolog is a complete depth-first traversal of the search tree, it was a natural choice to retain the top-down parsing strategy in our DCG facility as well. However, the general backtracking mechanism of Prolog contradicts the standard parsing principles in language processing: conventional DCGs parse the input program nondeterministically, while traditionally deterministic parsing is preferred. Nondeterministic parsing also torpedos syntax error handling since it makes hard to connect a recognized error to the erroneous grammar symbol. Moreover, nondeterministic parsing (although being a more general approach than deterministic one) is rarely actually needed in the context of programming languages because most programming languages are designed to be deterministically parsable.

Because of these reasons, we have based our DCG implementation on the context-free grammar class LL(*1*), i.e. parsing is a top-down left-to-right process

using a lookahead of length *l*. This choice makes our notation more restricted than the conventional one; the resulting formalism is rather related to one-pass attribute grammars or affix grammars [Kos 71] than to general attribute grammars.

Left recursion

Since our system can only process LL(*l*) grammars, left recursion is still forbidden. However, the system provides some relief in this restriction by automatically eliminating left-recursion from the original grammar, when asked. It also provides two other grammar transformations: left factoring, and elimination of useless productions. All these transformations have been implemented according to [ASU 86].

One shortness in these grammar transformations is that they are applied merely to the context-free part of the DCG; if the original grammar makes use of symbol arguments or procedure calls, these have to be updated on the transformed grammar by the user. The reason for excluding the semantic aspects from the DCG transformations is that a well-known result with attribute grammars shows that in general it is impossible to transform even an L-attributed grammar into an equivalent LL-attributed form [GiW 78] (preserving the level of semantic information during the transformation); thus an automatic semantic conversion would be doomed to failure.

Error recovery

Because we have based our implementation on deterministic parsing, we can employ standard syntax error handling techniques instead of just giving up, as is the case with the conventional DCG implementation. Our error recovery method is a combination of *panic mode* and *phrase-level* methods, as described in [WeM 80].

The idea is to always keep the parser 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 under parse. Synchronization is based on the FIRST and FOLLOW sets of nonterminals (see e.g. [ASU 86]).

The principle of error handling can be illustrated by giving as an example a procedure for parsing nonterminal A with production $A \rightarrow B$:

```
procedure A (Followers);
begin
    if not (Next in FIRST(A)) then begin
      Error ...;
      Skipto(FIRST(A)+Followers);
    end;
    if Next in FIRST(A) then begin
        B; — parse the right-hand side
        if not (Next in Followers) then begin
          Error ...;
          Skipto(Followers);
        end
    end
end.
```

Here the set Followers includes all the symbols in
$\text{FOLLOW}(A)+(\text{FOLLOW}(X_1)+\text{FOLLOW}(X_2)+\ldots+\text{FOLLOW}(X_n))$, $n>=0$,
where the symbols X_i represent the nonterminals on the path from A to the root in the underlying parse tree, i.e. all the nonterminals which have been entered but not yet exited. The $\text{FOLLOW}(X_i)$ sets guarantee that within any underlying parse tree a lower-level nonterminal cannot inadvertently skip over a token which a higher-level nonterminal expects to deal with.

Next represents the current input token, Error emits an appropriate error message, and Skipto(S) skips the input stream until a token in set S is found.

Because of the interactive nature of working with a Prolog interpreter, we have enriched this automatic form of recovery with the possibility for *local correction*: if requested, the parser always halts when detecting an error and asks the user to correct the current erroneous token. The available operations are replacement, insertion, and deletion.

We demonstrate the system by giving in the Appendix an example session.

4. Implementation

Our deterministic error-recovering DCG notation has been implemented using a meta-interpreter (see e.g. [StS 86]) that "interprets" the input grammar. Thus the solution is different from the conventional implementation where a DCG is first translated into ordinary Prolog and after that executed by a standard Prolog interpreter or compiler. The difference can be characterized more explicitly by sketching in Figures 1 and 2 the conventional implementation strategy and the metainterpreter strategy, respectively.

In our implementation the grammar is transformed into an internal representation of the DCG interpreter. This interpreter (a Prolog program) parses the source program by recursively applying a universal parser predicate with the current grammar symbol as parameter. The interpretation follows the principles discussed in chapter 3.

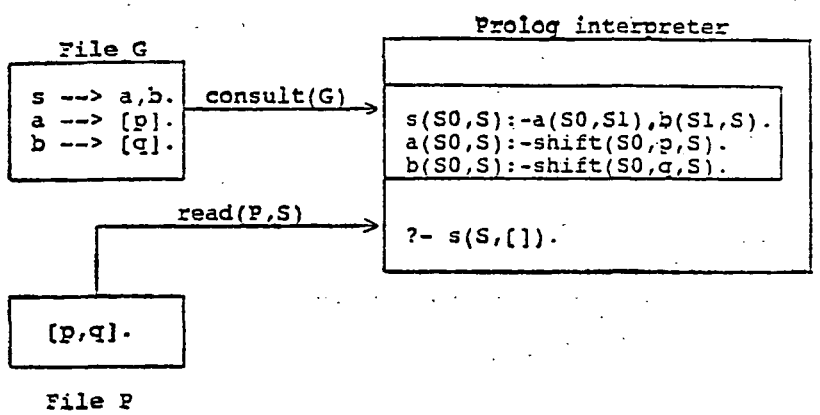

Figure 1. Conventional implementation of DCGs

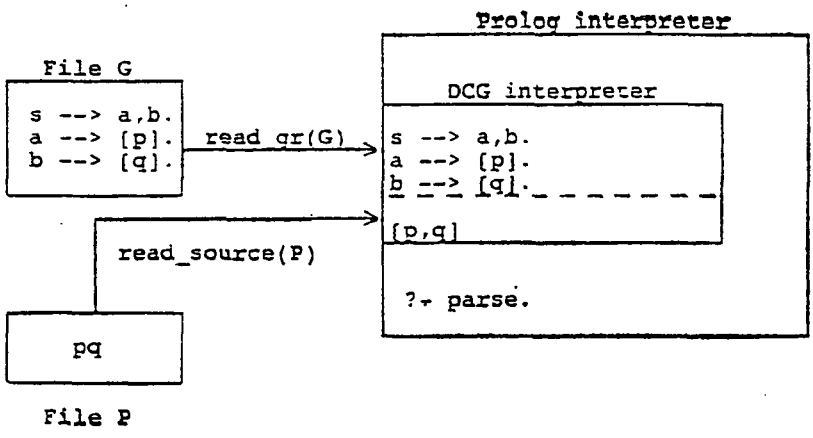

Figure 2. Meta-interpreter implementation of DCGs

In order to support lexical analysis, the system includes a standard scanner (read_source) that can be used for reading the source program and for converting it into a list of tokens. The lexical analyzer makes the conversion assuming "normal" patterns of "ordinary" token classes, such as identifiers, numbers, and operators. In case the lexical form of the source language does not match the assumptions made by the system, the user must either modify the standard analyzer or supply an analyzer of her/his own.

The system is embedded in Quintus Prolog [Qui 86], and it is described in more detail in [Top 89].

5. Experiences

Our DCG variant has been applied to several toy examples, such as arithmetic expressions. In these simple cases the system is superior to the conventional implementation: all the syntactic errors can be uncovered quite rapidly and even corrected on-the-fly. The automatic transformations free the user to some extent from artificial grammar constructions, such as right recursion.

Since the design of the system stems from practical problems with using Prolog for parsing, we have tested it in a more realistic case as well. The syntax of the programming language Edison [Bri 82] was specified as a DCG which was then executed both using our system and using Quintus Prolog. The efficiency of these parsers was analyzed and the results are given in Tables 1 and 2. The length of the source programs is indicated by lines, our system by Meta-DCG, and Quintus Prolog by Quintus-DCG. For Quintus Prolog we have assigned two figures, the first one being for the compiled parser and the second one for the interpreted parser. All the figures for our system are for the compiled parser. The tests have been carried out in a VAX/8800 under VMS.

Table. 1. Execution time of DCGs (seconds of cpu time)

lines	Quintus-DCG	Meta-DCG
10	0.01/0.1	7.8
60	0.06/0.7	88.4
100	0.08/0.8	101.8

Table 2. Memory consumption of DCGs (kbytes)

lines	Quintus-DCG	Meta-DCG
10	594/660	1368
60	654/899	4742
100	654/899	4875

As can be noticed, the meta-interpreter implementation unfortunately resulted in drastic loss of efficiency when compared to the conventional implementation by translation into ordinary Prolog. Even for relatively small Edison programs (less than 100 lines) the meta-interpreter was far too slow for practical consideration, and for source programs larger than 100 lines the Quintus Prolog system might run out of memory. Also parser initialization (loading the meta-interpreter, reading the DCG, checking the LL(*1*) property) took clearly more time than in the conventional case (reading the DCG, converting it into Quintus Prolog).

The main reason to this unfortunate inefficiency lies certainly in meta-interpretation. On one hand the program is quite complex and on the other hand the DCG is represented as data; thus no optimizations on the grammar can be done by the Prolog system as is the case with the conventional implementation. One part of the difference can be explained by the fact that our system has to check for syntactic correctness of the source program which task is totally outside the normal DCG model.

The primary goal of the system, automatic syntactic error recovery, has been reached to the extent that seems to be normal for this technique ([Har 77], [Pem 80]). The quality of error handling was analyzed by parsing syntactically erroneous Edison programs with the system. In ordinary cases the parser was able to find most of the actual errors, but on the other hand it reported quite many nonexistent errors (in some extreme cases the number of extraneous error messages was even larger than the number of actual error messages). When recovering from an error, the parser also skips some portion of the source program which in Edison's case is typically the whole incorrect structure (expression, statement, etc.). In the correction mode the amount of omitted text is usually smaller since user-supplied corrections can locally turn an invalid structure into a legal one.

6. Discussion and Future Work

This work shares some of the contributions with previous research on parsing and Prolog. Deterministic parsing with Prolog based on LL(*1*) grammars is discussed in [Abr 86], some systems circumvent the problems with left-recursion by employing bottom-up parsing (e.g. BUP [MTK 86], AID [Nil 86]), etc. However, as far as we

know the automatic error handling mechanism is unique in our system. Also the DCG transformations (albeit merely on the context-free part of the grammar) are something new. We emphasize the methodological aspect in our system; of course the same tasks could be carried out by the user as well (we have produced yet another Edison parser as a DCG with explicit error handling [Paa 89]) but that would significantly lower the conceptual level of the DCG notation.

Restricting the implementation on LL(*l*) grammars with FIRST and FOLLOW sets imposes some problems compared with the conventional implementation (besides reducing the set of accepted grammars). In our DCG variant it is not sensible to make use of terminal variables, as in

number → [C], {is_number(C)}.

This would include variable C in FIRST(number), and the consequence would be that a syntactically erroneous number symbol would not be detected by the parser (since each possible token *t* would be considered valid through unification C=*t*). Another problem of similar nature is that a grammar with the following alternative productions is not LL(*l*) in our sense:

factor → [C], {is_number(C)}.
factor → [id].

The reason to this is that the sets FIRST([C]) and FIRST([id]) are not considered disjoint (again since C always unifies with id). A general solution to these problems is hard to find. In both example cases we could and actually should use procedure is_number to generate all the possible ground patterns for C and make use of this pattern set instead of C in computing the FIRST and FOLLOW sets, but in general such lexical auxiliary procedures are rather hard to automatically locate in a DCG.

One interesting problem to be solved in the future is to integrate lexical analysis with parsing in DCGs. As noted in chapter 2, the traditional DCG formalism does not support such an integration, and we also have excluded it from our implementation. Another topic for the future is to base parsing and error recovery on the translation from DCG into ordinary Prolog, as is done in conventional implementations. This strategy would certainly be more efficient than our current one: besides that meta-interpretation as the implementation method was shown to be rather inefficient, conceptually the relation between a translation-based implementation and the meta-interpreter-based implementation clearly bears an analogy to the relation between (faster) parser programs and (slower) table-driven parsers. In the translation mode it would also be easier for the user to correct a non-LL(*l*) grammar or to retain the semantics during context-free transformations, since all the implementation-dependent information (such as the FIRST and FOLLOW sets) that is currently hidden within the meta-interpreter would be explicitly available in terms of Prolog.

Acknowledgements. We appreciate Prof. Esko Ukkonen's participation in a number of fruitful discussions on the topic.

References

[Abr 86] ABRAMSON H.: Sequential and Current Deterministic Logic Grammars. In: Proc. of the 3rd International Conference on Logic Programming, London, 1986. Lecture Notes in Computer Science 225, Springer-Verlag, 1986, 389-395.

[ASU 86] AHO A. V., SETHI R., ULLMAN J. D.: Compilers — Principles, Techniques and Tools. Addison-Wesley, 1986.

[Bri 82] BRINCH HANSEN P.: Programming a Personal Computer. Prentice-Hall, 1982.

[Col 73] COLMERAUER A.: Les systemes-Q ou un Formalisme pour Analyser et Synthesizer des Phrases sur Ordinateur. Publication Interne No. 43, Dept. d'Informatique, Universite de Montreal, 1973.

[GiW 78] GIEGERICH R., WILHELM R.: Counter-One-Pass Features in One-Pass Compilation — A Formalization Using Attribute Grammars. Information Processing Letters 7, 6, 1978, 279—284.

[Har 77] HARTMAN A. C.: A Concurrent Pascal Compiler for Minicomputers. Lecture Notes in Computer Science 50, Springer-Verlag, 1977.

[Knu 68] KNUTH D. E.: Semantics of Context-Free Languages. Mathematical Systems Theory 2, 2, 1968, 127—145.

[Kos 71] KOSTER C. H. A.: Affix Grammars. In: Peck J. E. L.: Algol 68 Implementation, North-Holland, 1971, 95—109.

[MTK 86] MATSUMOTO Y., TANAKA H., KIYONO M.: BUP: A Bottom-Up Parsing System for Natural Languages. In: Logic Programming and Its Applications (van Caneghem M., Warren D., eds.), Ablex Publishing Co., 1986.

[Nil 86] NILSSON U.: Alternative Implementation of DCGs. New Generation Computing 4, 4, 1986, 383—399.

[Paa 89] PAAKKI J.: Comparison of Compiler Writing Methods: An Experiment. In: Proc. of the 13th Information Technologies Conference, Sarajevo, 1989. Science and Research Council of Bosnia and Hertzegovina, 1989, R 122.

[Pem 80] PEMBERTON S.: Comments on an Error-Recovery Scheme by Hartmann. Software Practice and Experimence 10, 3, 1980, 231—240.

[PeW 80] 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.

[Qui 86] Quintus Computer Systems, Inc: Quintus Prolog Reference Manual, Version 6, 1986.

[StS 86] STERLING L., SHAPIRO S.: The Art of Prolog. The MIT Press, 1986.

[Top 89] TOPPOLA K.: An Error-Recovering DCG Feature (in Finnish). Report C—1989—23, Department of Computer Science, University of Helsinki, 1989.

[WeM 80] WELSH J., MCKEAG M.: Structured System Programming. Prentice-Hall, 1980.

Appendix

An example session, starting with automatic error recovering and finishing with user-supplied local correction. The commands by the user are given in **bold**.

? — **consult(dcg).**
yes
? — **read_grammar(g1).**
The grammar is not LL(1).
yes
?— **list_grammar.**
(1) expr ---> expr,"+",term.
(2) expr ---> term.
(3) term ---> term, "*",factor.
(4) term ---> factor.

```
 (5) factor ----> "(",expr,")".
 (6) factor ----> "id".
yes
? — transform(e).
Eliminating left recursion ...
yes
? — list_grammar.
 (1) expr ----> "(",expr,")",term1,expr1.
 (2) expr ----> "id",term1,expr1.
 (3) term ----> "(",expr,")",term1.
 (4) term ----> "id",term1.
 (5) factor ----> "(",expr,")".
 (6) factor ----> "id".
 (7) expr1 ----> "+",term,expr1.
 (8) expr1 ----> "".
 (9) term1 ----> "*",factor,term1.
(10) term1 ----> "".
yes
? — parse("id+id").
Parsing completed, 0 errors detected.
yes
? — parse("id*(id—id)+id").
id*(id
--- Error 1 ---
Unexpected symbol(s) met and skipped:
— id
Parsing completed, 1 errors detected.
yes
? — correction(on).
yes
? — parse("id*(id—id)+id")
id*(id
--- Error 1 ---
** _ **
One of the following expected:
* + )
Replace(r)/insert(i)/delete(d) token:
==> r(+).
Parsing completed, 1 errors detected.
yes
```