Definite Clause Grammars with Parse Trees: Extension for Prolog

Falco Nogatz
University of Würzburg, Department of Computer Science,
Am Hubland, 97074 Würzburg, Germany
falco.nogatz@uni-wuerzburg.de

Dietmar Seipel
University of Würzburg, Department of Computer Science,
Am Hubland, 97074 Würzburg, Germany
dietmar.seipel@uni-wuerzburg.de

Salvador Abreu
LISP, Department of Computer Science, University of Évora, Portugal
spa@uevora.pt

Abstract

Definite Clause Grammars (DCGs) are a convenient way to specify possibly non-context-free grammars for natural and formal languages. They can be used to progressively build a parse tree as grammar rules are applied by providing an extra argument in the DCG rule's head. In the simplest way, this is a structure that contains the name of the used nonterminal. This extension of a DCG has been proposed for natural language processing in the past and can be done automatically in Prolog using term expansion.

We extend this approach by a meta-nonterminal to specify optional and sequences of nonterminals, as these structures are common in grammars for formal, domain-specific languages. We specify a term expansion that represents these sequences as lists while preserving the grammar's ability to be used both for parsing and serialising, i.e. to create a parse tree by a given source code and vice-versa. We show that this mechanism can be used to lift grammars specified in extended Backus–Naur form (EBNF) to generate parse trees. As a case study, we present a parser for the Prolog programming language itself based only on the grammars given in the ISO Prolog standard which produces corresponding parse trees.

2012 ACM Subject Classification Theory of computation → Grammars and context-free languages

Keywords and phrases Definite Clause Grammar, Prolog, Term Expansion, Parse Tree, EBNF

Digital Object Identifier 10.4230/OASIcs.SLATE.2019.7

Supplement Material The source codes of *library(dcg4pt)* and *library(plammar)* are available on GitHub at `https://github.com/fnogatz/dcg4pt` and `https://github.com/fnogatz/plammar` (MIT License).

1 Introduction

The Logic Programming language Prolog has a long history in natural language parsing, and so have *Definite Clause Grammars* (DCGs). Since its introduction by Alain Colmerauer in the early 1970s [6], Prolog has been developed with a focus on *natural language processing* (NLP). This led to *Metamorphosis Grammars* [5] in 1978, a first framework based on first-order logic to parse French. Its rewriting rule mechanism led to the development of DCGs in 1980 [11]. Unlike the established extended Backus–Naur form (EBNF), DCGs come with logical variables and brought all of Prolog's built-in capabilities to drive the parsing process, they could therefore also be used to describe non-context-free grammars.

© Falco Nogatz, Dietmar Seipel, and Salvador Abreu;
licensed under Creative Commons License CC-BY
8th Symposium on Languages, Applications and Technologies (SLATE 2019).
Editors: Ricardo Rodrigues, Jan Janoušek, Luís Ferreira, Luísa Coheur, Fernando Batista, and Hugo Gonçalo Oliveira; Article No. 7; pp. 7:1–7:14

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Besides their use for natural languages, DCGs are the means of choice for parsing and serialising formal languages and data exchange formats in Prolog. For instance, SWI-Prolog's *library(http/html_write)*, which is used to parse and generate HTML fragments, relies on DCGs. A new application to describe formal languages using DCGs arose with the introduction of quasi-quotations [16] in SWI-Prolog. This feature allows the embedding of any external domain-specific language (DSL) directly within Prolog code. For instance, Nogatz et al. used this approach with quasi-quotations to add support for *GraphQL* to SWI-Prolog [9]. The DSL is parsed by a DCG at compile time and replaced by the generated parse tree. It is likely that the embedding of external DSLs in Prolog will become even more popular with the integration of quasi-quotations, increasing the need for a mechanism that creates the corresponding Prolog term for the embedded string based on the DSL's grammar.

The remainder of the paper is organised as follows. In Section 2, we introduce the working with grammars in Prolog. As most formal languages are specified in EBNF, we compare this notation with DCGs, Prolog's de-facto standard for grammars. Our proposed modified term expansion is introduced in Section 3. As an example application, we present in Section 4 a parser for Prolog source code which uses a generative grammar that is based on the definitions in the ISO Prolog standard. In Section 5, we present existing extensions to the DCG formalism and argue about the relative merits of this approach when compared to others. Finally, we conclude with a summary and outlook of future applications in Section 6.

2 Grammars in Prolog

In this section, we first shortly introduce the extended Backus–Naur form, which is the de-facto standard to describe formal languages, including ISO Prolog. We then present the syntax and semantics of DCGs in Prolog as well as their usage and translation in SWI-Prolog.

2.1 Extended Backus–Naur Form

EBNF is a notation to formally describe grammars with production rules. The list of EBNF rules consists of nonterminals and symbols (terminals). Symbols are typically alphanumeric characters, punctuation marks, etc., specified in quotation marks.

Each rule has three parts: a *left-hand side* (LHS) of just a single nonterminal, a *right-hand side* (RHS) consisting of nonterminals and symbols, and the = symbol, which separates the two sides and reads as "is defined as". The elements of the RHS either describe an ordered sequence (denoted by commas ,) or alternative choices (denoted by vertical bars |, with a smaller precedence than the ordered sequence). Repetitions are enclosed by curly brackets { ... }, optional nonterminals by square brackets [...], and comments by brackets of the form (∗ ... ∗).

As a motivational example in this paper, we consider in Figure 1 an extract of the ISO Prolog standard that specifies the syntax of a variable token [1, Sec. 6]. A Prolog variable is either the anonymous variable given by the underscore character, or a named variable which has to start with an underscore character or capital letter. For instance, *"_"* is the anonymous variable, and *"_a"* and *"A"* are named variables. The comments refer to the sections of the ISO Prolog standard where the nonterminals are defined.

2.2 Syntax of DCGs

DCGs are not yet part of the ISO Prolog standard [1], but are under consideration for inclusion in the future [2]. Nevertheless, as of today DCGs are supported by all major Prolog implementations, including SWI-Prolog [14].

variable token	=	*anonymous variable* (∗ 6.4.3 ∗)
	|	*named variable* (∗ 6.4.3 ∗) ;
anonymous variable	=	*variable indicator char* (∗ 6.4.3 ∗) ;
named variable	=	*variable indicator char* (∗ 6.4.3 ∗),
		alphanumeric char (∗ 6.5.2 ∗),
		{ *alphanumeric char* (∗ 6.5.2 ∗) }
	|	*capital letter char* (∗ 6.5.2 ∗),
		{ *alphanumeric char* (∗ 6.5.2 ∗) } ;
variable indicator char	=	*underscore char* (∗ 6.5.2 ∗) ;
underscore char	=	"_" ;
capital letter char	=	"A" | "B" | "C" | . . . ;
alphanumeric char	=	. . .

Figure 1 EBNF rules for a variable token in Prolog.

In its simplest form to just specify context-free grammars, DCGs are very similar to EBNF. Again, a DCG is a list of *grammar rules*, consisting of a LHS and a RHS. Instead of =, the Prolog operator `-->/2` is used in between.[1] For compatibility with EBNF, the vertical bar `|/2` can be used to denote alternatives, but Prolog's traditional disjunction `;/2` is also supported.

Compared to EBNF, DCGs provide three major extensions, resulting in the ability to describe possibly non-context-free grammars:

- *Arguments in the LHS.* In contrast to EBNF, the nonterminal on the LHS of a DCG is allowed to have any number of arguments of any type. Since it is common in Prolog to use the same variables for input and output, these additional arguments can also be used to describe the corresponding parse tree, that either is used as input while serialising or gets generated while parsing.
- *Complex control structures in the RHS.* Besides the *conjunction* `,/2` and the *disjunction* `|/2` and `;/2`, the Prolog control structures for *If-Then/-Else* `->/2` and *not* `\+/1` can be used to express relationships between items in the RHS. *Parentheses* `(...)` and Prolog's *cut* operator `!/0` can be used as usual. In addition, any Prolog code can be embedded by using curly brackets `{...}`.
- *Pushback arguments.* DCGs allow the definition of rules in the form "`H, P --> B1, ..., Bn`", with `P` being a list of terminals that are prepended to the parsed list after successfully evaluating the grammar's body [2]. As we do not use it in our application, it is mentioned here only for completeness.

The arguments in the LHS are also often used together with embedded Prolog code in the RHS to condition the application of a rule or alternative by some provided options. For instance, as of version 7.3.27 SWI-Prolog provides a flag `var_prefix` [15, Sec. 2.16.1] that

[1] In the rest of this paper, we will use the notation `A/N` to denote a Prolog operator with the name `A` and an arity of `N`. In contrast, a nonterminal `A` with `N` arguments is denoted by `A//N`.

restricts the syntax of variables. Given `var_prefix(true)`, only variables starting with the underscore character are allowed, i.e. identifiers starting with a capital letter denote atoms.[2]

We can reproduce this behaviour by extending the DCG's `named_variable` LHS by the argument `Flags` and put the condition to check for `var_prefix(false)` in front of the nonterminals on the RHS:

```
named_variable(Flags) -->
  { option(var_prefix(false), Flags) },
  capital_letter_char, ...
```

As usually in Prolog, comments are written as `/* ... */`. Unlike EBNF, DCG provides no pre-defined syntax neither for optional nonterminals nor repetitions.

2.3 EBNF as an Internal DSL in Prolog

EBNF and DCG are already very similar in their syntax. In fact, the example of Figure 1 can be embedded directly into Prolog with only minor modifications:

- Nonterminals in DCGs must be valid Prolog atoms, so included whitespaces have to be replaced, e.g., by underscore characters.
- Comments are written as `/* ... */` instead of `(* ... *)`.[3]
- The very last rule have to end with a dot `"."`.

Because SWI-Prolog and YAP allow the definition of the *block operators* `[]/1` and `{}/1` [15, Sec. 5.3.3], this slightly modified EBNF is already valid Prolog syntax. However, to not confuse them with Prolog's list notation and DCG's embedded Prolog code, we write optional elements as `?c` instead of `[ c ]`, and sequences as `*c` instead of `{ c }`, with `?/1` and `*/1` defined as prefix operators.

As of SWI-Prolog version 7, text enclosed in double quotes is read as objects of type *string*. Using the Prolog directive `:- set_prolog_flag(double_quotes, chars).` this can be changed, so that double-quoted text is read as a list of characters. As a result, terminals in DCGs can be written as strings enclosed in double quotes as in EBNF.

Prolog provides a mechanism to rewrite Prolog code at compilation time, similar to macros in other programming languages. This is called *term expansion*. When loading code into SWI-Prolog, its compiler calls the predicate `expand_term/2` on each term read from the input. As part of it, `term_expansion/2` is executed to apply user-defined term expansions first. With term expansion, EBNF can be translated into normal DCG notation at compile time. For instance, the single EBNF rule *underscore_char* = "_" gets replaced by the following Prolog fact:

```
user:term_expansion(A = B, [A --> B]).
```

With similar expansions for "|" (alternatives), ";" (rule endings), "?" (optional elements), and " * " (sequences), EBNF becomes an internal DSL in Prolog. As a result, grammars of formal languages provided as EBNF can be directly used in Prolog, resulting in executable parsers.

[2] This has been introduced for compatibility with Prolog by BIM. It has proven to be useful for defining internal domain-specific languages in Prolog that require identifiers to start with an uppercase letters, e.g., for RDF.

[3] Note that ehe term `(* ... *)` is valid Prolog syntax when `*/1` is defined as both a prefix and postfix operator, since every Prolog term is allowed to be bracketed. However, this would require a comma in front of the comment, since a bracketed term is only allowed as an argument.

2.4 Procedural Semantics of DCGs and its Implementation

Essentially, the DCG notation is only syntactic sugar for writing Prolog predicates that operate on difference lists. In most Prolog implementations, DCGs are translated into normal Prolog code at compilation time using term expansion and the pre-defined predicate `dcg_translate_rule/2`. It adds the two arguments which are normally hidden by the DCG notation.

For instance, consider the definition of the second alternative for named variables of Figure 1 that describes a named variable beginning with a capital letter. It is only allowed for the `var_prefix` flag set to `false` (cf. Section 2.2). Using appropriate term expansions as presented in Section 2.3, the equivalent DCG is as follows:

```
named_variable(Flags) -->
  { option(var_prefix(false), Flags) },
  capital_letter_char, /* 6.5.2 */
  *alphanumeric_char.  /* 6.5.2 */
```

For every nonterminal `A//N`, two additional arguments `S` and `R` are added, resulting in a Prolog predicate `A/(N+2)` with `S = [C|R]`, and `C` the list of symbols consumed by the rule's body. Every body item operates on the result of the previous one. If there is embedded Prolog code given in curly brackets, it is inserted at the specified position. I.e., the aforementioned DCG for `named_variable//1` gets expanded to the Prolog predicate `named_variable/3`:

```
named_variable(Flags, A, C) :-
  option(var_prefix(false), Flags),
  capital_letter_char(A, B),
  *(alphanumeric_char, B, C).
```

The generated Prolog predicates can be directly used. For instance, the following call consumes all possible prefixes of the string "*Abc.D*" that are valid named variables and returns the rest:

```
?- Flags=[var_prefix(false)], L="Abc.D", named_variable(Flags,L,R).
R = "bc.D" ; % first solution
R = "c.D" ;  % second solution
R = ".D" .   % last solution because "." is no alphanumeric character
```

Prolog backtracks over the three possibilities for the sequence of `alphanumeric_char//0`, beginning with the empty string. Following Prolog's *SLD resolution* mechanism, the rules are tried in their order of appearance. With Prolog's *backtracking* mechanism, multiple rules with compatible LHSs will be tried. For the rest of the paper, we assume a basic knowledge of these two fundamentals of Prolog.

3 Modified Term Expansion for Parse Tree Generation

The practical benefits of the DCG presented in Section 2.4 are very limited – the grammar can only be used to check if a given string can be parsed by the grammar. Even generating all allowed variable names is narrow, because Prolog's SLD resolution first backtracks over the sequence of `alphanumeric_char//0`, i.e. it generates the possible variable names in the order of "*Aa*", "*Aaa*", "*Aaaa*" instead of "*Aa*", "*Ab*", "*Ac*".

For practical use, the application of the grammar shall generate the corresponding parse tree on-the-fly. In the field of natural language processing, it has been proposed to extend the DCG's LHS by an additional argument that holds the parse tree. Following the ideas proposed

by Abramson and Dahl [3], the additional parse tree argument for a DCG rule `H --> B` is a term of the form `H(T)`, with `H` being the name of the rule's head without arguments, and `T` a term whose structure depends on the rule's body `B`. This way, `named_variable//1` becomes `named_variable//2`:

```
named_variable(Flags, named_variable(T1,T2)) -->
  { option(var_prefix(false), Flags) },
  capital_letter_char(T1), *(alphanumeric_char, T2).
```

This construction method is generic because the additional parse tree argument is constructed based only on the LHS's nonterminal symbol and the structure of the grammar rule's RHS. The extension of an existing DCG can therefore be done automatically at compile time using term expansion.

As part of our contribution, we provide a Swi-Prolog package *library(dcg4pt)* ("DCG for parse trees") which defines a predicate `dcg4pt_rule_to_dcg_rule/2` that takes a DCG as its first argument and returns an equivalent DCG where the nonterminals have been extended by an additional parse tree argument. The library is listed in Swi-Prolog's package list at `http://www.swi-prolog.org/pack/list?p=dcg4pt`. Its source code is published under MIT License at `https://github.com/fnogatz/dcg4pt`. It can be used to get the extended version of every DCG rule at first, and translate the result afterwards using Prolog's built-in predicate `dcg_translate_rule/2` as introduced in Section 2.4:

```
:- use_module(library(dcg4pt)).
user:term_expansion(H --> B, Rule) :-
  dcg4pt_rule_to_dcg_rule(H --> B, DCG),
  dcg_translate_rule(DCG, Rule).
```

3.1 Handling of Optionals and Sequences of Nonterminals

Grammars for formal languages often make great use of optional and sequences of nonterminals. For instance, the nonterminal *named variable* of Figure 1 allows any number of alphanumeric characters, including zero. Another typical use case is the whitespace for indentation in programming languages; newline and whitespace characters can be set arbitrarily and even include comments. This is different from the previous applications in the field of natural language processing as in [3]. There, the number of nonterminals in a grammar rule's RHS is known in advance. As a result, the parse tree argument can have a fixed number of children. E.g., the parse tree is represented as `np(Name)` for a noun phrase that is simply a name; or `np(Det,Noun,Rel)` for a noun phrase that consists of a determiner, noun, and relative clause.

However, when working with optionals and sequences, the number of children is not limited. It is therefore desirable to use a list if there is a conjunction, optional, or sequence on a grammar rule's RHS. The DCG for `named_variable//2` should therefore produce a parse tree of the form `named_variable([T1|T2])`, with `T1` being the parse tree generated by `capital_letter_char//1`, and `T2` the (possibly empty) list of parse trees each generated by `alphanumeric_char//1` in the sequence.

This follows the ideas of the library of [12], which also introduces additional control structures for parsing sequences, such as `sequence(Mode,...)`, that further make the code more compact, readable and declarative, where `Mode` can be one of `'*'`, `'**'`, `'+'`, and `'?'`. The previous library is the basis of our current implementation of *library(dcg4pt)*. It is available in the *Declare* package at `http://www1.pub.informatik.uni-wuerzburg.de/databases/ddbase/` and has been used in an application of DCGs to language processing

for electronic dictionaries in linguistics by Seipel et al. [12], and represents the parse trees in an XML term for Prolog. Instead of using `sequence/2`, *library(dcgt4pt)* defines the prefix operators `?/1` (optional element), `+/1` (non-empty sequence), and `*/1` (possibly empty sequence).

3.2 Term Expansion Scheme

The general formation principles of *library(dcg4pt)* depending on the body of a DCG rule are presented in Table 1. The parse tree is always added as the very last argument, i.e. for a rule's LHS of `h(some)`, the head of the generated DCG rule becomes `h(some, h(...))`.

Table 1 Formation principles to construct the parse tree for a DCG rule "`h --> Body`".

DCG Body	Example DCG	Extended by Parse Tree Argument
Terminal	`h --> "_".`	`h(h('_')) --> "_".`
Nonterminal	`h --> a.`	`h(h(A)) --> a(A).`

Conjunction	`h --> a , b.`	`h(h(R0)) -->`
		`{ R0 = [A|R1] }, a(A),`
		`{ R1 = [B] }, b(B).`

Disjunction	`h --> a | b.`	`h(h(R0)) -->`
		`{ R0 = A }, a(A) ;`
		`{ R0 = B }, b(B).`

Embedded Prolog	`h --> a, { p }.` *Embedded Prolog is ignored for the parse tree generation.*	`h(h(A)) --> a(A), { p }.`
Sequence or Optional	`h --> *a.` *And similar for the prefix operators +/1 and ?/1.* *R is a list.* *In `sequence//3`, we distinguish whether the DCG is called with bound or unbound arguments.*	`h(h(R)) --> sequence('*',a,R).`

Note that the embedded Prolog code for variable unifications presented in Table 1 is needed to support complex RHSs with combinations of all these structures. For instance, a rule "`h --> a, *b, c`" should not be translated into:

```
1 h(h([A,Bs,C])) --> a(A), sequence('*',b,Bs), c(C).
```

Because `sequence//3` describes a list `Bs`, the generated parse tree for `h//1` would otherwise contain a list of lists. Instead, the following extended DCG is generated by *library(dcg4pt)*:

```
1 h(h(R0)) -->
2   { R0=[A|R1] }, a(A),
3   sequence('*', b, Bs), { append(Bs, R2, R1) },
4   { R2=[C] }, c(C).
```

With no `b//1` being present in the sequence, the resulting parse tree for `h//1` is just `h([A,C])`.

Using this compilation scheme, the extended DCG is capable to generate all possible combinations of strings and corresponding parse trees. For instance, in Listing 1, `variable_token/3` is used to parse an input of `"_a"`. Using backtracking, it returns two solutions. First, only the first symbol `"_"` is consumed, because it represents the anonymous

variable; the remainder "a" of the string is bound to the third argument. As a second solution, the string is parsed as a named variable.

Listing 1 Usage example for the generated `variable_token/3` with the input string `"_a"`.

```
?- variable_token(T, "_a", R).
% first solution, consuming only "_":
R = "a", T = variable_token(anonymous_variable(
             variable_indicator_char(underscore_char('_')))) ;
% second solution, consuming the whole string "_a":
R = "",  T = variable_token(named_variable([
             variable_indicator_char(underscore_char('_')),
             alphanumeric_char(alpha_char(letter_char(
             small_letter_char(a) ))) ])) .
```

3.3 Support for Parsing and Serialising

The aim of *library(dcg4pt)*'s term expansion scheme is to create a modified DCG that expresses a relation between the string and parse tree. In particular, the generated Prolog program can also be used "in reverse" to a normal parser, i.e. to serialise a string by a given parse tree. For this purpose it has to be ensured that the term expansion scheme presented in Section 3.2 uses only Prolog predicates that are *pure*, i.e. they can be used no matter which of the arguments are bound. For instance, the aforementioned rule "`h --> a, *b, c`" could also be expanded to use Prolog's built-in predicate `flatten/2`. It calculates a flattened list from a list of lists and therefore also avoids the use of nested lists. However, `flatten(+ListOfLists,-FlattenedList)` can not be used the other way around. As a result, the generated extendend DCG can only be used to parse a given string and return the corresponding parse tree; serialising a given parse tree back to the corresponding string is not possible.

Note that this is an improvement on the previous implementations of [7] and [12]. In addition, because of possibly left-recursive rules, or rules that consume resp. produce no symbols, the expanded rules have to behave differently depending on whether they are called with bound or unbound arguments. For instance, consider the rules that describe a *variable* as presented in Listing 2: it is a *variable_token* optionally prepended by layout characters. *layout_text_sequence* succeeds for any non-empty sequence of whitespace, tab, or newline characters.

Listing 2 Grammar rules for the nonterminal *variable* in Prolog.

```
variable = ?layout_text_sequence, variable_token ;
layout_text_sequence = layout_text, *layout_text ;
layout_text = layout_char | comment ;
layout_char = space_char | horizontal_tab_char | new_line_char .
```

For a given string `"  _a"` (the string `"_a"` preceded by two whitespaces), the RHS of *layout_text_sequence* should try to consume as many whitespace characters as possible to avoid unnecessary backtracking. On the other hand, this greedy approach is undesirable when both arguments are unbound, i.e. when generating all allowed strings with their corresponding parse tree. In that case, the smallest possible string should be created at first. The four combinations of an (un)bound string or parse tree are automatically handled by the DCGs generated by our tool. For instance, in the definition of `*//1` (resp. `*/4` after expansion with the arguments for the parse tree, and the lists `S` and `R`) in Listing 3, we test whether

the argument `S` is a variable. If so, we make use of `sequence('*',_)` that starts with the smallest string, i.e. the empty list, whereas `sequence('**',_)` tries to consume as many symbols as possible from the given argument `S`. Similar checks have been implemented for the sequences `'?'` and `'+'`, resulting in Prolog programs that can be used both for parsing and serialising, based on a single grammar.

Listing 3 Implementation of the meta-predicate `*/4` to support sequences of nonterminals.

```
*(DCGBody, Tree, S, R) :-
  \+ var(S), !,
  sequence('**', DCGBody, Tree, S, R).
*(DCGBody, Tree, S, R) :-
  var(S), !,
  sequence('*', DCGBody, Tree, S, R).
```

4 Case Study: A Parser for Prolog

Quite often parsing is only a single step when working with formal languages. A common use case is program transformations. These require to first parse the program based on a grammar, then generate an abstract syntax tree, modify it, and serialise it again. With our tool, the same language specification – i.e. the same code – can be used for the parsing and serialising steps. They share a single data structure – the parse tree which was automatically added by our tool's modified term expansion –, and the resulting Prolog program can be used in both directions without any modification. Compared to the common approach of using a parser generator like ANTLR [10] instead, our library relieves the programmer from the burden of keeping two tools, for parsing and serialising, in sync.

As a case study, we present the implementation of a grammar for Prolog programs. With the help of our tool *library(dcg4pt)*, the DCG is extended by a hidden argument to store the parse tree. We will show how this generic parse tree can be modified and used by the same grammar to produce a valid Prolog program again. It can be used in SWI-Prolog as the package *library(plammar)*.

The programming language Prolog is specified in the ISO Prolog standard [1]. While most of the syntax is described using EBNF, the ISO standard also contains informally specified requirements which cannot be expressed by context-free-grammars. For instance, it provides grammar rules for the language's tokens, but also states informally:

> *A token shall not be followed by characters such that concatenating the characters of the token with these characters forms a valid token [...].*

This requirement cannot be expressed by a context-free grammar. As the ISO standard contains several similar requirements, parsing Prolog is a prime example for a realistic parser based on DCGs.

Analysing the syntax of a programming language usually requires two phases: (i) the lexical analysis, that converts a sequence of characters into a sequence of tokens, and (ii) the parsing of the tokens in order to generate a structural representation. With DCGs it is possible to write *scannerless parsers* that combine these two steps into a single grammar. However, the ISO standard defines Prolog similarly: it first declares that a Prolog program consists of Prolog terms that are a sequence of tokens, and later defines the grammars for tokens and terms separately. Therefore, our implementation is also split into the two phases. Both make use of grammars but work on lists of different types: the *lexer* handles the program source code as a string and is presented in Section 4.1; the *parser* works with a

list of tokens and is described in Section 4.2. An example on how to use this grammar for a source-to-source transformation is presented in Section 4.3. In Section 4.4 we present the integration of *library(plammar)* into a graphical interface to interactively explore a parse tree.

4.1 Lexical Analysis

The ISO standard specifies the syntax of Prolog in more than 200 EBNF grammar rules. After having defined EBNF as an internal DSL as presented in Section 2.3, the grammar rules can be directly used as a Prolog program.

The beginning of the DCGs as defined in the ISO standard is given in Listing 4. It states that a *term* is a sequence of *token*. A *token* is one of *name*, *variable*, *integer*, *float_number*, etc.

Listing 4 Definition of tokens according to Sec. 6.4 of the ISO Prolog standard [2].

```
1 term = *token ;  % sequence of token
2 token = name | variable | integer | float_number | ht_sep | open |
3           close | open_ct | double_quoted_list | comma | open_list |
4           close_list | open_curly | close_curly .
```

As presented in Section 3, these grammar rules are expanded so that they match the corresponding parse tree. For *token* this is simply a structure `token(I)` with `I` one of `name(...)`, `variable(...)`, and so on, because every element of the choice is a single nonterminal. The EBNF in the ISO standard is deeply structured and thus creates very verbose parse trees. For instance, consider the Prolog term that consists of just the named variable "`_a`" as given as the second solution in Listing 1.

The extended DCG for the nonterminal *term* describes a parse tree of the form `term(I)`. According to our modified term expansion, `I` is always a list of `token(...)`. This is exactly the result one expects from a lexer.

As an example, we consider the implementation of the `member/2` predicate in Prolog:

```
1 member(X,[X|_]).
2 member(X,[_|Xs]) :-
3   member(X,Xs).
```

`member(?Elem,?List)` is true, iff `Elem` is the `List`'s first element (fact in l. 1), or one of the following (recursive rule in ll. 2). The tokenisation of the first line of this implementation is given in Listing 5. Our tool *library(plammar)* provides the Prolog predicate `prolog_tokens(?Source,?Tokens)` that takes Prolog source code and generates the list of tokens, and the other way around. For the `member/1` fact it returns 11 tokens.

Listing 5 Tokenisation of "`member(X,[X|_]).`" as generated by *library(plammar)*.

```
1  ?- use_module(library(plammar)). % load package
2  ?- prolog_tokens(string("member(X,[X|_])."), Tokens).
3  Tokens = [
4    name([name_token(letter_digit_token([small_letter_char(m),...]))]),
5    open_ct(open_token(open_char('('))),
6    variable([variable_token(named_variable([capital_letter_char('X')]))]),
7    comma([comma_token(comma_char(','))]),
8    open_list([open_list_token(open_list_char('['))]),
9    variable(/* as before for X */),
10   ht_sep([head_tail_separator_token(/* shortened */)]),
11   variable([variable_token(anonymous_variable(/* as in Listing 1 */))]),
12   close_list([close_list_token(close_list_char(']'))]),
13   close([close_token(close_char(')'))]),
14   name([name_token(graphic_token([graphic_token_char(graphic_char(.))]))
15 ]) ] .
```

4.2 Parsing

DCGs are not only useful when working with strings. In general, they describe difference lists over any type, and strings are just a special case, since in Prolog they are represented by a list of characters. Thus, the grammar to process the list of tokens generated in the lexical analysis can also be defined by an extended DCG. The ISO standard even specifies valid sequences of tokens as EBNF, as well. For instance, the compound term `member(X,[X|_])` is a valid Prolog term: it is the sequence of an *atom*, the opening parenthesis, a list of arguments (denoted by the nonterminal *arg_list*), and the closing parenthesis. The corresponding grammar rule is presented in Listing 6. Here, elements given in brackets represent terminals, i.e. the parse trees `open_ct(...)` and `close(...)` are elements in the list of tokens generated by the lexer. Using our modified term expansion, this rule describes a parse tree of the form `term([atom(...), open_ct(...), arg_list(...), close(...)])`.

Listing 6 Definition of compound terms in functional notation according to Sec. 6.3.3 of the ISO Prolog standard [2].

```
1 term(0) = atom, [ open_ct(_) ], arg_list, [ close(_) ] ;
2 atom = [ name(_) ] ;
3 arg_list = arg ;
4 arg_list = arg, [ comma(_) ], arg_list ;
5 arg = term(P), { P < 1000 } .
```

Note that parsing Prolog source code requires one to annotate all terms by their precedence (called *priority* in the ISO standard). A compound term has the highest precedence *zero*, which is specified in the argument of *term*. The nonterminal *term* therefore has a higher arity than the *term* of Listing 4, making the two predicates distinguishable: our term expansion creates a Prolog predicate `term/4` for the former, and `term/3` for the latter.

Given the definitions of Listing 6, the tokens returned by `prolog_tokens/2` (cf. Listing 5) are correctly recognised as a Prolog term in functional notation. The term consists of a function symbol that is an atom – represented by the list element `name(_)` –, followed by the opening parenthesis `open_ct(_)`, the list of arguments *arg_list*, and the closing parenthesis `close(_)`. *arg_list* is a comma-separated list of terms with a precedence lower than 1000. The token `variable(_)` is used as the first argument. The second argument given by the tokens `[open_list(_), variable(_), ht_sep(_), variable(_), close_list(_)]` denotes a list and is parsed by another grammar rule for `term(0)`, which we do not present in detail here.

4.3 Source-to-Source Transformation

The Prolog grammar provided in *library(plammar)* can be used for parsing and serialising. An application would be code reformatting. For instance, let's assume we want to automatically add whitespace characters after every comma in the *arg_list* of a compound term. Given a Prolog predicate `transform_parse_tree/2`, we can apply the source-to-source transformation by just combining the created `term/3` and `term/4` predicates, as they can be used in both directions:

```
1 ?- term(term(ListOfTokens_1), Old, []),        % lexical analysis
2    term(Prec, AST_1, ListOfTokens_1, []),      % parsing
3    transform_parse_tree(AST_1, AST_2),          % transformation
4    term(Prec, AST_2, ListOfTokens_2, []),      % serialise tokens
5    term(term(ListOfTokens_2), New, []).        % serialise string
```

In *library(plammar)*, we provide Prolog predicates to ease the work with tokens, parse trees, and abstract syntax trees: `prolog_parsetree/2`, `parsetree_ast/2`, and `prolog_ast/2`. They have been developed with a focus on their usage as relations and can handle the input of only the Prolog source code, only the tokens, parse tree, or abstract syntax tree, or both arguments being unbound.

4.4 Integration into Graphical AST Explorer

As part of our contribution we integrated the Prolog parser *library(plammar)* into `https://astexplorer.net/`, an open source web application that provides parsers for several programming languages, including PHP, JavaScript, and SQL. Figure 2 presents the graphical representation of the generated parse tree for the Prolog program that defines the `member/2` predicate.

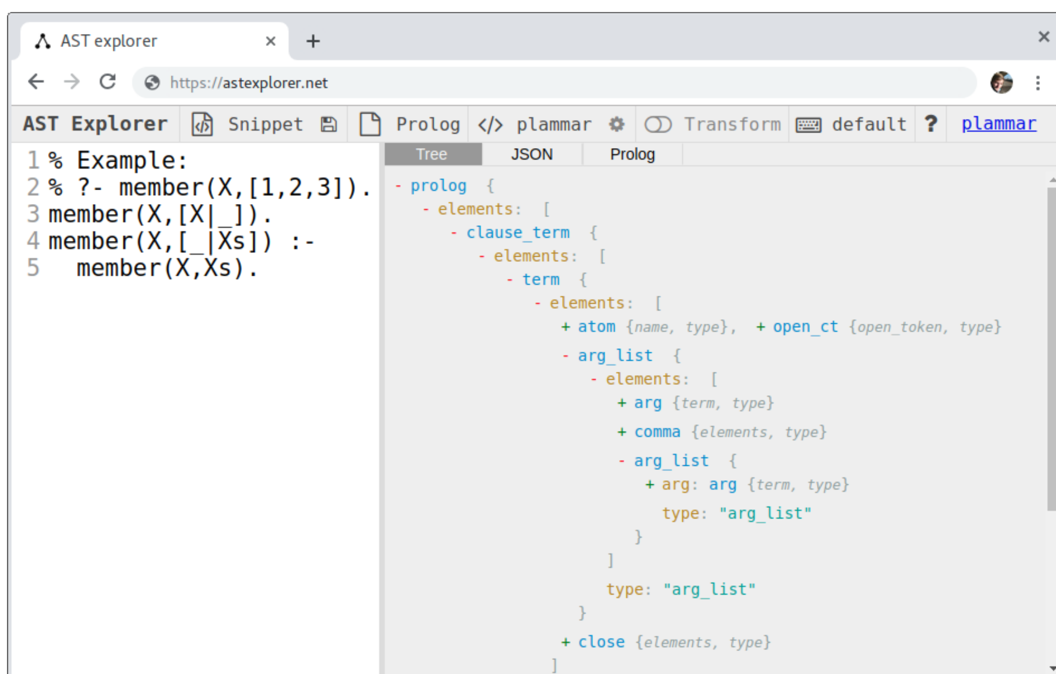

Figure 2 Integration of the Prolog parser into `https://astexplorer.net/`.

5 Related Work

Since its introduction by A. Colmerauer, the logic programming language Prolog was developed with a focus on natural language processing. This resulted in a first representation of grammars as clauses of first-order logic in 1975 by Colmerauer [4, 5]. Definite Clause Grammars were introduced by Pereira and Warren in 1980 [11]. As a usage example of extra arguments in nonterminals, they manually extend rules that parse sentences by their corresponding *building structures* – a term holding information about the applied rule and its RHS elements.

This idea is adopted by Dahl and McCord in 1983 [7]. Their *modifier structure grammars* extend a grammar with two additional arguments to obtain a meaning representation (called *semantic structure*), and its corresponding *syntactic structure* in form of a parse tree. Simultaneously and independently, *restriction grammars* were developed by Hirschman and

Puder [8], which also automatically create parse trees. An overview of these approaches is given in [3, Chapters 7–8], where the idea of hiding the parse tree argument from the user is discussed.

The aforementioned approaches are focused on context-free grammars. In particular, they do not make use of embedded Prolog on a rule's RHS, and higher-order structures like sequences. Although they expand grammar rules by an additional argument to store a parse tree, its actual construction is not specified. Hence, we have observed that they do not address the challenges that arise when grammar rules that consume resp. produce no symbols are called with unbound arguments. This is a requirement for grammars that are to be used for both parsing and serialising.

In [13], DCGs are again augmented with hidden additional arguments. Instead of generating parse trees, they allow to define multiple accumulators, e.g., to calculate and store the size of the consumed symbols. Accumulators are defined using Prolog predicates. It might be possible to use this technique to define a hidden accumulator that creates the corresponding parse tree, though to the best of our knowledge this has not yet been done.

6 Conclusion

The development of Definite Clause Grammars was long driven by a focus on natural language processing. As of today, other techniques have gained popularity in this area. In this paper, we emphasise the usefulness of DCGs for the work with formal languages, as DCGs provide a unified mean to specify both a parser and serialiser. We took up again the idea of extending DCGs by a hidden argument to store the corresponding parse tree, and presented a generalised formation principle that supports optional and sequences of nonterminals and provides optimisations depending on which arguments are bound. Our tool can be used with any existing DCG, resulting in a generative grammar. It is published as the Swi-Prolog package *library(dcg4pt)*.[4]

As an example application, we implemented a generative grammar for Prolog source code, bundled as *library(plammar)*.[5] Because Prolog's syntax is defined by more than 200 grammar rules in EBNF, we implemented EBNF as an internal domain-specific language for Prolog. This enabled an easy adoption of the ISO Prolog standard with only minor modifications.

The tool for automatic parse tree generation will become useful for implementing external domain-specific languages in Prolog, in particular using quasi-quotations. On the other hand, there are several future applications for the Prolog parser: besides source-to-source transformation for refactoring, it can be used for static source code analysis, e.g., to find all predicates that are available only in some Prolog systems.

References

1 ISO/IEC 13211-1:1995: Information technology – Programming languages – Prolog – Part 1: General core, 1995.

2 ISO/IEC 13211-3:2006: Definite clause grammar rules, 2015.

3 Harvey Abramson and Veronica Dahl. *Logic Grammars. Symbolic Computation.* Springer, 1989.

4 Alain Colmerauer. Les grammaires de métamorphose. Technical report, Groupe d'Intelligence Artificielle, Université de Marseille-Luminy, 1975.

[4] `https://github.com/fnogatz/dcg4pt` (MIT License)
[5] `https://github.com/fnogatz/plammar` (MIT License)

5. Alain Colmerauer. Metamorphosis grammars. In *Natural language communication with computers*, pages 133–188. Springer, 1978.
6. Alain Colmerauer. An Introduction to Prolog III. *Communications of the ACM*, 33(7):69–90, 1990. doi:10.1145/79204.79210.
7. Veronica Dahl and Michael C. McCord. Treating coordination in logic grammars. *American Journal of Computational Linguistics*, 9(2):69–80, 1983.
8. Lynette Hirschman and Karl Puder. Restriction grammar: A Prolog implementation. *Logic Programming and its Applications*, pages 244–261, 1985.
9. Falco Nogatz and Dietmar Seipel. Implementing GraphQL as a Query Language for Deductive Databases in SWI–Prolog Using DCGs, Quasi Quotations, and Dicts. In *Proc. 30th Workshop on Logic Programming (WLP)*, 2016.
10. Terence J. Parr and Russell W. Quong. ANTLR: A predicated-LL (k) parser generator. *Software: Practice and Experience*, 25(7):789–810, 1995.
11. Fernando Pereira and David Warren. Definite clause grammars for language analysis – a survey of the formalism and a comparison with augmented transition networks. *Artificial intelligence*, 13(3):231–278, 1980.
12. Christian Schneiker, Dietmar Seipel, Werner Wegstein, and Klaus Prätor. Declarative Parsing and Annotation of Electronic Dictionaries. In *6th International Workshop on Natural Language Processing and Cognitive Science (NLPCS)*, pages 122–132, 2009.
13. Peter Van Roy. Extended DCG notation: A tool for Applicative Programming in Prolog. Technical Report UCB/CSD 90/583, Computer Science Division, UC Berkeley, 1990.
14. Jan Wielemaker. An Overview of the SWI-Prolog Programming Environment. In *Proc. 13th International Workshop on Logic Programming Environments (WLPE)*, pages 1–16, 2003.
15. Jan Wielemaker. SWI-Prolog Reference Manual 7.6, 2017.
16. Jan Wielemaker and Michael Hendricks. Why It's Nice to be Quoted: Quasiquoting for Prolog. In *Proc. 23rd Workshop on Logic-based Methods in Programming Environments (WLPE)*, 2013.