A Simple Transformation for Offline-Parsable Grammars and its Termination Properties

Marc Dymetman*
Rank Xerox Research Centre
6, chemin de Maupertuis
Meylan, 38240 France
dymetman@xerox.fr

Abstract We present, in easily reproducible terms, a simple transformation for offline-parsable grammars which results in a provably terminating parsing program directly top-down interpretable in Prolog. The transformation consists in two steps: (1) removal of empty-productions, followed by: (2) left-recursion elimination. It is related both to left-corner parsing (where the grammar is compiled, rather than interpreted through a parsing program, and with the advantage of guaranteed termination in the presence of empty productions) and to the Generalized Greibach Normal Form for DCGs (with the advantage of implementation simplicity).

1 Motivation

Definite clause grammars (DCGs) are one of the simplest and most widely used unification grammar formalisms. They represent a direct augmentation of context-free grammars through the use of (term) unification (a fact that tends to be masked by their usual presentation based on the programming language Prolog). It is obviously important to ask wether certain usual methods and algorithms pertaining to CFGs can be adapted to DCGs, and this general question informs much of the work concerning DCGs, as well as more complex unification grammar formalisms (to cite only a few areas: Earley parsing, LR parsing, left-corner parsing, Greibach Normal Form).

One essential complication when trying to generalize CFG methods to the DCG domain lies in the fact that, whereas the parsing problem for CFGs is decidable, the corresponding problem for DCGs is in general undecidable. This can be shown easily as a consequence of the noteworthy fact that any definite clause *program* can be viewed as a definite clause *grammar* "on the empty string", that is, as a DCG where no terminals other than [ ] are allowed on the right-hand sides of rules. The Turing-completeness of definite clause programs therefore implies the undecidability of the parsing problem for this subclass of DCGs, and *a fortiori* for DCGs in general.[1] In order to guarantee good computational properties for DCGs, it is then necessary to impose certain restrictions on their form such as *offline-parsability* (OP), a nomenclature introduced by Pereira and Warren [11], who define an OP DCG as a grammar whose context-free skeleton CFG is not infinitely ambiguous, and show that OP DCGs lead to a decidable parsing problem.[2]

Our aim in this paper is to propose a *simple* transformation for an arbitrary OP DCG putting it into a form which leads to the completeness of the direct top-down interpretation by the standard Prolog interpreter: parsing is guaranteed to enumerate all solutions to the parsing problem and terminate. The existence of such a transformation is known: in [1, 2], we have recently introduced a "Generalized Greibach Normal Form" (GGNF) for DCGs, which leads to termination of top-down interpretation in the OP case. However, the available presentation of the GGNF transformation is rather complex (it involves an algebraic study of the fixpoints of certain equational systems representing grammars.). Our aim here is to present a related, but much simpler, transformation, which from a theoretical viewpoint performs somewhat less than the GGNF transformation (it involves some encoding of the initial DCG, which the GGNF does not, and it only handles offline-parsable grammars, while the GGNF is defined for arbitrary DCGs),[3] but in practice is extremely easy to implement and displays a comparable behavior when parsing with an OP grammar.

The transformation consists of two steps: (1) empty-production elimination and (2) left-recursion elimination.

The empty-production elimination algorithm is inspired by the usual procedure for context-free grammars. But there are some notable differences, due to the fact that removal of empty-productions is in general impossible for non-OP DCGs. The empty-

*Thanks to Pierre Isabelle and Franois Perrault for their comments, and to CITI (Montreal) for its support during the preparation of this paper.

[1] DCGs on the empty string might be dismissed as extreme, but they are in fact at the core of the offline-parsability concept. See note 3.

[2] The concept of offline-parsability (under a different name) goes back to [8], where it is shown to be linguistically relevant.

[3] The GGNF factorizes an arbitrary DCG into two components: a "unit sub-DCG on the empty string", and another part consisting of rules whose right-hand side starts with a terminal. The decidability of the DCG depends exclusively on certain simple textual properties of the unit sub-DCG. This sub-DCG can be eliminated from the GGNF if and only if the DCG is offline-parsable.

production elimination algorithm is guaranteed to terminate only in the OP case.[4] It produces a DCG declaratively equivalent to the original grammar.

The left-recursion elimination algorithm is adapted from a transformation proposed in [4] in the context of a certain formalism ("Lexical Grammars") which we presented as a possible basis for building reversible grammars.[5] The key observation (in slightly different terms) was that, in a DCG, if a nonterminal g is defined literally by the two rules (the first of which is left-recursive):

$$g(X) \rightarrow g(Y),\ d(Y,X).$$
$$g(X) \rightarrow t(X).$$

then the replacement of these two rules by the three rules (where d_tc is a new nonterminal symbol, which represents a kind of "transitive closure" of d):

$$g(X) \rightarrow t(Y),\ d\_tc(Y,X).$$
$$d\_tc(X,X) \rightarrow [\,].$$
$$d\_tc(X,Z) \rightarrow d(X,Y),\ d\_tc(Y,Z).$$

preserves the declarative semantics of the grammar.[6]

We remarked in [4] that this transformation "is closely related to left-corner parsing", but did not give details. In a recent paper [7], Mark Johnson introduces "a left-corner program transformation for natural language parsing", which has some similarity to the above transformation, but which is applied to definite clause programs, rather than to DCGs. He proves that this transformation respects declarative equivalence, and also shows, using a model-theoretic approach, the close connection of his transformation with left-corner parsing [12, 9, 10].[7]

It must be noted that the left-recursion elimination procedure can be applied to any DCG, whether OP or not. Even in the case where the grammar is OP, however, it will *not* lead to a terminating parsing algorithm *unless* empty productions have been prealably eliminated from the grammar, a problem which is shared by the usual left-corner parser-interpreter.

[4]The fact that the standard CFG empty-production elimination transformation is always possible is related to the fact that this transformation does not preserve degrees of ambiguity. For instance the infinitely ambiguous grammar $S \rightarrow [b]\ A,\ A \rightarrow A,\ A \rightarrow [\,]$ is simplified into the grammar $S \rightarrow [b]$. This type of simplification is generally impossible in a DCG. Consider for instance the "grammar" $s(X) \rightarrow [number]\ a(X),\ a(succ(X)) \rightarrow a(X),\ a(0) \rightarrow [\,]$.

[5]The method goes back to a transformation used to compile out certain local cases of left-recursion from DCGs in the context of the Machine Translation prototype CRITTER [3].

[6]A proof of this fact, based on a comparison of proof-trees for the original and the transformed grammar, is given in [2].

[7] His paper does not state termination conditions for the transformed program. Such termination conditions would probably involve some generalized notion of offline-parsability [6, 5, 13]. By contrast, we prove termination only for DCGs which are OP in the original sense of Pereira and Warren, but this case seems to us to represent much of the core issue, and to lead to some direct extensions. For instance, the DCG transformation proposed here can be directly applied to "guided" programs in the sense of [4].

Due to the space available, we do not give here correctness proofs for the algorithms presented, but expect to publish them in a fuller version of this paper. These algorithms have actually been implemented in a slightly extended version, where they are also used to decide whether the grammar proposed for transformation is in fact offline-parsable or not.

2 Empty-production elimination

It can be proven that, if DCG0 is an OP DCG, the following transformation, which involves repeated partial evaluation of rules that rewrite into the empty string, terminates after a finite number of steps and produces a grammar DCG without empty-productions which is equivalent to the initial grammar on non-empty strings:[8]

input: an offline-parsable DCG1.
output: a DCG without empty rules equivalent to DCG1 on non-empty strings.
algorithm:
initialize LIST1 to a list of the rules of DCG1, set LIST2 to the empty list.
while there exists an empty rule ER: $A(T1,...,Tk) \rightarrow [\,]$ in LIST1 **do:**
move ER to LIST2.
for each rule R: $B(...) \rightarrow \alpha$ in LIST1 such that α contains an instance of $A(...)$ (including new such rules created inside this loop) **do:**
for each such instance $A(S1,...,Sk)$ unifiable with $A(T1,...,Tk)$ **do:**
append to LIST1 a rule R': $B(...) \rightarrow \alpha'$ obtained from R by removing $A(S1,...,Sk)$ from α (or by replacing it with $[\,]$ if this was the only nonterminal in α), and by unifying the Ti's with the Si's.
set DCG to LIST1.

For instance the grammar consisting in the nine rules appearing above the separation in fig. 1 is transformed into the grammar (see figure):

$$s(s(NP,VP)) \rightarrow np(NP), vp(VP).$$
$$np(np(N,C)) \rightarrow n(N), comp(C).$$
$$n(n(people)) \rightarrow [people].$$
$$vp(vp(v(sleep),C)) \rightarrow [sleep], comp(C).$$
$$comp(c(C,A)) \rightarrow comp(C), adv(A).$$
$$adv(adv(here)) \rightarrow [here].$$
$$adv(adv(today)) \rightarrow [today].$$
$$np(np(n(you)),C) \rightarrow comp(C).$$
$$np(np(N,nil)) \rightarrow n(N).$$
$$comp(c(nil,A)) \rightarrow adv(A).$$
$$vp(vp(v(sleep),nil)) \rightarrow [sleep].$$
$$s(s(np(np(n(you)),nil),VP)) \rightarrow vp(VP).$$

[8]When DCG0 is not OP, the transformation may produce an infinite number of rules, but a simple extension of the algorithm can detect this situation: the transformation stops and the grammar is declared not to be OP.

3 Left-recursion elimination

The transformation can be logically divided into two steps: (1) an encoding of DCG into a "generic" form DCG', and (2) a simple replacement of a certain group of left-recursive rules in DCG' by a certain equivalent non left-recursive group of rules, yielding a top-down interpretable DCG". An example of the transformation DCG $\longrightarrow$ DCG' $\longrightarrow$ DCG" is given in fig. 2.

The encoding is performed by the following algorithm:

input: an offline-parsable DCG without empty rules.
output: an equivalent "encoding" DCG'.
algorithm:
initialize LIST to a list of the rules of DCG.
initialize DCG' to the list of rules (literally):
$g(X) \rightarrow g(Y),\ d(Y,X).$
$g(X) \rightarrow t(X).$
while there exists a rule R of the form
$A(T1,...,Tk) \rightarrow B(S1,...,Sl)\ \alpha$ in LIST **do:**
remove R from LIST.
add to DCG' a rule R':
$d(B(S1,...,Sl), A(T1,...,Tk)) \rightarrow \alpha'$,
where α' is obtained by replacing any $C(V1,...,Vm)$
in α by $g(C(V1,...,Vm))$,
or is set to [ ] in the case where α is empty.
while there exists a rule R of the form
$A(T1,...,Tk) \rightarrow [terminal]\ \alpha$ in LIST **do:**
remove R from LIST.
add to DCG' a rule R':
$t(A(T1,...,Tk)) \rightarrow [terminal]\ \alpha'$,
where α' is obtained by replacing any $C(V1,...,Vm)$
in α by $g(C(V1,...,Vm))$,
or is set to [ ] in the case where α is empty.

The procedure is very simple. It involves the creation of a generic nonterminal $g(X)$, of arity one, which performs a task equivalent to the original nonterminals $s(X1,\ldots,Xn), vp(X1,\ldots,Xm), ...$ The goal $g(s(X1,\ldots,Xn))$, for instance, plays the same role for parsing a sentence as did the goal $s(X1,\ldots,Xn)$ in the original grammar.

Two further generic nonterminals are introduced: $t(X)$ accounts for rules whose right-hand side begins with a terminal, while $d(Y,X)$ accounts for rules whose right-hand side begins with a nonterminal. The rationale behind the encoding is best understood from the following examples, where $\Longrightarrow$ represents rule rewriting:

$vp(vp(v(sleep),C)) \rightarrow [sleep],\ comp(C)$
$\Longrightarrow g(vp(vp(v(sleep),C))) \rightarrow [sleep],\ g(comp(C))$
$\Longrightarrow g(X) \rightarrow [sleep],$
$\underbrace{(\ \{X = vp(vp(v(sleep),C))\},\ g(comp(C))\ )}_{t(X)}$

$s(s(NP,VP)) \rightarrow np(NP),\ vp(VP)$
$\Longrightarrow g(s(s(NP,VP))) \rightarrow g(np(NP)),\ g(vp(VP))$
$\Longrightarrow g(X) \rightarrow g(Y),$
$\underbrace{(\ \{X = s(s(NP,VP)),\ Y = np(NP)\},\ g(vp(VP))\ )}_{d(Y,X)}$

The second example illustrates the role played by $d(Y,X)$ in the encoding. This nonterminal has the following interpretation: X is an"immediate" extension of Y using the given rule. In other words, Y corresponds to an "immediate left-corner" of X.

The left-recursion elimination is now performed by the following "algorithm":[9]

input: a DCG' encoded as above.
output: an equivalent non left-recursive DCG".
algorithm:
initialize DCG" to DCG'.
in DCG", replace literally the rules:
$g(X) \rightarrow g(Y),\ d(Y,X).$
$g(X) \rightarrow t(X).$
by the rules:
$g(X) \rightarrow t(Y),\ d\_tc(Y,X).$
$d\_tc(X,X) \rightarrow [\ ].$
$d\_tc(X,Z) \rightarrow d(X,Y),\ d\_tc(Y,Z).$

In this transformation, the new nonterminal d_tc plays the role of a kind of transitive closure of d. It can be seen that, relative to DCG", for any string w and for any ground term z, the fact that $g(z)$ rewrites into w —or, equivalently, that there exists a ground term x such that $t(x)\ d\_tc(x,z)$ rewrites into w— is equivalent to the existence of a sequence of ground terms $x = x_1,\ \ldots,\ x_k = z$ and a sequence of strings $w_1,\ \ldots,\ w_k$ such that $t(x_1)$ rewrites into w_1, $d(x_1,x_2)$ rewrites into w_2, ..., $d(x_{k-1},x_k)$ rewrites into w_k, and such that w is the string concatenation $w = w_1 \cdots w_k$. From our previous remark on the meaning of $d(Y,X)$, this can be interpreted as saying that "consituent x is a left-corner of constituent z", relatively to string w.

The grammar DCG" can now be compiled in the standard way—via the adjunction of two "differential list" arguments—into a Prolog program which can be executed directly. If we started from an offline-parsable grammar DCG0, this program will enumerate all solutions to the parsing problem and terminate after a finite number of steps.[10]

References

[1] Marc Dymetman. A Generalized Greibach Normal Form for Definite Clause Grammars. In *Proceedings of the 15th International Conference on*

[9] In practice, this and the preceding algorithm, which are dissociated here for exposition reasons, are lumped together.

[10] There exist of course DCGs which do not contain empty productions and which are *not* OP. They are characterized by the existence of cycles of "chain-rules" of the form: $a_1(...) \rightarrow a_2(...)$. , ..., $a_{m-1}(...) \rightarrow a_m(...)$. , with $a_m = a_1$. But, if we start with an OP DCG0, the empty-production elimination algorithm cannot produce such a situation.

LIST1	delete	LIST2
$s(s(NP,VP)) \rightarrow np(NP), vp(VP).$		
$np(np(N,C)) \rightarrow n(N), comp(C).$		
$n(n(people)) \rightarrow [people].$		
$n(n(you)) \rightarrow [\,].$	$\times$	
$vp(vp(v(sleep),C)) \rightarrow [sleep], comp(C).$		
$comp(c(C,A)) \rightarrow comp(C), adv(A).$		
$comp(nil) \rightarrow [\,].$	$\times$	
$adv(adv(here)) \rightarrow [here].$		
$adv(adv(today)) \rightarrow [today].$		
		$n(n(you)) \rightarrow [\,].$
$np(np(n(you)),C) \rightarrow comp(C).$		
		$comp(nil) \rightarrow [\,].$
$np(np(N,nil)) \rightarrow n(N).$		
$comp(c(nil,A)) \rightarrow adv(A).$		
$vp(vp(v(sleep),nil)) \rightarrow [sleep].$		
$np(np(n(you)),nil) \rightarrow [\,].$	$\times$	
		$np(np(n(you)),nil) \rightarrow [\,].$
$s(s(np(np(n(you)),nil),VP)) \rightarrow vp(VP).$		

Figure 1: Empty-production elimination.

Computational Linguistics, volume 1, pages 366–372, Nantes, France, July 1992.

[2] Marc Dymetman. Transformations de grammaires logiques et réversibilité en Traduction Automatique. Thèse d'Etat, 1992. Université Joseph Fourier (Grenoble 1), Grenoble, France.

[3] Marc Dymetman and Pierre Isabelle. Reversible logic grammars for machine translation. In *Proceedings of the Second International Conference on Theoretical and Methodological Issues in Machine Translation of Natural Languages*, Pittsburgh, PA, June 1988. Carnegie Mellon University.

[4] Marc Dymetman, Pierre Isabelle, and François Perrault. A symmetrical approach to parsing and generation. In *Proceedings of the 13th International Conference on Computational Linguistics*, volume 3, pages 90–96, Helsinki, August 1990.

[5] Andrew Haas. A generalization of the offline-parsable grammars. In *Proceedings of the 27th Annual Meeting of the Association for Computational Linguistics*, pages 237–42, Vancouver, BC, Canada, June 1989.

[6] Mark Johnson. *Attribute-Value Logic and the Theory of Grammar*. CSLI lecture note No. 16. Center for the Study of Language and Information, Stanford, CA, 1988.

[7] Mark Johnson. A left-corner program transformation for natural language parsing, (forthcoming).

[8] R. Kaplan and J. Bresnan. Lexical functional grammar: a formal system for grammatical representation. In Bresnan, editor, *The Mental Representation of Grammatical Relations*, pages 173–281. MIT Press, Cambridge, MA, 1982.

[9] Y. Matsumoto, H. Tanaka, H. Hirikawa, H. Miyoshi, and H. Yasukawa. BUP: a bottom-up parser embedded in Prolog. *New Generation Computing*, 1(2):145–158, 1983.

[10] Fernando C. N. Pereira and Stuart M. Shieber. *Prolog and Natural Language Analysis*. CSLI lecture note No. 10. Center for the Study of Language and Information, Stanford, CA, 1987.

[11] Fernando C. N. Pereira and David H. D. Warren. Parsing as deduction. In *Proceedings of the 21th Annual Meeting of the Association for Computational Linguistics*, pages 137–144, MIT, Cambridge, MA, June 1983.

[12] D. J. Rosencrantz and P. M. Lewis. Deterministic left-corner parsing. In *Eleventh Annual Symposium on Switching and Automata Theory*, pages 139–153. IEEE, 1970. Extended Abstract.

[13] Stuart M. Shieber. *Constraint-Based Grammar Formalisms*. MIT Press, Cambridge, MA, 1992.

DCG

$s(s(NP,VP)) \rightarrow np(NP), vp(VP).$
$np(np(N,C)) \rightarrow n(N), comp(C).$
$n(n(people)) \rightarrow [people].$
$vp(vp(v(sleep),C)) \rightarrow [sleep], comp(C).$
$comp(c(C,A)) \rightarrow comp(C), adv(A).$
$adv(adv(here)) \rightarrow [here].$
$adv(adv(today)) \rightarrow [today].$
$np(np(n(you)),C) \rightarrow comp(C).$
$np(np(N,nil)) \rightarrow n(N).$
$comp(c(nil,A)) \rightarrow adv(A).$
$vp(vp(v(sleep),nil)) \rightarrow [sleep].$
$s(s(np(np(n(you)),nil),VP)) \rightarrow vp(VP).$

DCG'

$g(X) \rightarrow g(Y), d(Y,X).$
$g(X) \rightarrow t(X).$
$d(np(NP), s(s(NP,VP))) \rightarrow g(vp(VP)).$
$d(n(N), np(np(N,C))) \rightarrow g(comp(C)).$
$t(n(n(people))) \rightarrow [people].$
$t(vp(vp(v(sleep),C))) \rightarrow [sleep], g(comp(C)).$
$d(comp(C), comp(c(C,A))) \rightarrow g(adv(A)).$
$t(adv(adv(here))) \rightarrow [here].$
$t(adv(adv(today))) \rightarrow [today].$
$d(comp(C), np(np(n(you)),C)) \rightarrow [\,].$
$d(n(N), np(np(N,nil))) \rightarrow [\,].$
$d(adv(A), comp(c(nil,A))) \rightarrow [\,].$
$t(vp(vp(v(sleep),nil))) \rightarrow [sleep].$
$d(vp(VP), s(s(np(np(n(you)),nil),VP))) \rightarrow [\,].$

DCG"

$g(X) \rightarrow t(Y), d\_tc(Y,X).$
$d\_tc(X,X) \rightarrow [\,].$
$d\_tc(X,Z) \rightarrow d(X,Y), d\_tc(Y,Z).$
$d(np(NP), s(s(NP,VP))) \rightarrow g(vp(VP)).$
$d(n(N), np(np(N,C))) \rightarrow g(comp(C)).$
$t(n(n(people))) \rightarrow [people].$
$t(vp(vp(v(sleep),C))) \rightarrow [sleep], g(comp(C)).$
$d(comp(C), comp(c(C,A))) \rightarrow g(adv(A)).$
$t(adv(adv(here))) \rightarrow [here].$
$t(adv(adv(today))) \rightarrow [today].$
$d(comp(C), np(np(n(you)),C)) \rightarrow [\,].$
$d(n(N), np(np(N,nil))) \rightarrow [\,].$
$d(adv(A), comp(c(nil,A))) \rightarrow [\,].$
$t(vp(vp(v(sleep),nil))) \rightarrow [sleep].$
$d(vp(VP), s(s(np(np(n(you)),nil),VP))) \rightarrow [\,].$

Figure 2: Encoding (DCG') of a grammar (DCG) and left-recursion elimination (DCG").