Adaptable Grammars for Non-Context-Free Languages

Extended and revised version[1]

Henning Christiansen
Research group PLIS: Programming, Logic and Intelligent Systems
Department of Communication, Business and Information Technologies
Roskilde University, P.O.Box 260, DK-4000 Roskilde, Denmark
henning@ruc.dk, http://www.ruc.dk/~henning

Abstract

We consider, as an alternative to traditional approaches for describing non-context-free languages, the use of grammars in which application of grammar rules themselves control the creation or modification of grammar rules. This principle is shown to capture, in a concise way, standard example languages that are considered as prototype representatives of non-context-free phenomena in natural languages. We define a grammar formalism with these characteristics and show how it can be implemented in logic programming in a surprisingly straightforward way, compared with the expressive power. It is also shown how such adaptable grammars can be applied for describing meta-level architectures that include their own explicit meta-languages for defining new syntax.

1 Introduction

An important aspect of formal linguistics is to search for suitable grammar formalisms that can describe non-context-free languages; both the formal expressibility as well as *how* these languages are described, are important (are the grammars "natural" in some sense, or which conception of language does a class of grammars represent).

Many classical approaches to handle extra-context-free aspects apply specialized derivation relations so that everything is captured in terms of rewriting; two-level grammars [22] and contextual grammars [17] are examples of this. Other directions use extra-grammatical add-on's to context-free grammars such

[1]This paper is an extended and revised version of [7] presented at *Third International Workshop on Non-Classical Formal Languages in Linguistics*, organized by Gemma Bel-Enguix and M. Dolores Jiménez-López, published previously in the IWANN 2009 Proceedings, LNCS, vol. 5517.

as first order logic in Definite Clause Grammars [20] and arbitrary mathematical functions in Attribute Grammars [15].

We consider here a third approach by turning the grammar itself into a dynamic entity, so that the application of grammar rules can create new or modify existing grammar rules. The scope of grammar rules can be controlled analogously to the way Definite Clause Grammars and Attribute Grammars pass information around to different parts of the string being analyzed or constructed. The difference is that here we pass the entire and dynamically evolving grammar around, so that new constructs may come in and others disappear.

We present a formalism that we call Adaptable Grammars, based on an earlier proposal of ours [3, 4, 5], which has not been considered in a formal linguistics setting before, but confined to applications for programming languages.

The principle of having a grammar dynamically modifying or adapting itself along a discourse may be seen as a universal mechanism that can be incorporated in other grammatical frameworks as well. One of the motivations for this approach is to obtain a 'natural' way of modelling those context-dependent aspects of language that essentially are related to the introduction of new linguistic potential or – in a broader perspective – the development of language, whether this be within a discourse or perhaps over generations of language users or even in the evolution of species. The advantage is that original and novel language constructs are represented in an equal manner so that, at any stage, the current grammar may be read out. In most traditional grammar formalisms, that are capable of expressing some context-dependencies, this need to be modelled by highly over-general rules whose application is controlled by an encoding of the linguistic context.

The earliest version of our grammars were introduced with the misleading name of 'generative grammars' that was later changed into 'adaptable grammars'; other authors have used the name 'Christiansen grammars', e.g., [21, 13, 19, 24, 9]. Recently [19, 9] have used these grammars for grammatical, evolutionary programming; the authors motivate their approach by the observation that with such grammars, they can do with shorter derivations of target programs, thus smaller and more expressive 'chromosomes' and faster convergence.

The first trace of the principle of grammars that adapt to the context seems to be [11] from the early 1960ies, but no general formalism was developed; an extensible grammar formalism was developed in [23] for describing one particular system, and other proposals are [21, 1]; see also [4, 5] for a detailed comparison with other selected formalism.

In the present paper, we define in section 2 an adaptable grammar formalism which borrows formal concepts from first-order logic and logic programming. However, we have intended to make the paper accessible also for readers with little or no background in the logic programming field. In section 3 we show a generic implementation written as a meta-interpreter in Prolog, which is strikingly simple compared with the high expressibility of the class of adaptable grammars that it implements; this section may be skipped by mainly linguistically oriented readers and summarized as "there is an implementation that can be copied from this paper, pasted into any standard Prolog system and executed directly". We show how these grammars can capture standard non-context-free languages used in the literature (e.g., [18]) as prototypical representatives for the central natural language properties of *reduplication*, *crossed dependencies*, and *multiple agreements* (section 4; find executable versions in the appendix). Finally, in section 5, we show how adaptable grammars can be used to describe meta-level architectures that include their own explicit meta-languages for defining new syntax. It appears that the three first grammars become almost trivial when formulated as adaptable grammars, whereas the last example requires a more detailed analysis analysis of the input and synthesis of new grammar rules. The concluding section gives a summary and a discussion of perspectives and possible future work.

2 Definition

We assume the terminology of first-order logic, including logical variables, and logic programs (pure Prolog programs) as well as related notions such as (ground) instances for formulas and terms; see, e.g., [16]. We recall briefly that a ground term is one without logical variables, and that a ground instance of a formula or term is given by a consistent replacement of ground terms for its variables. Intuitively, a ground formula represents a specific piece of information whereas a non-ground one can be instantiated in many different way; for example $\mathit{identical}(X, Y)$, where X and Y are variables, may represent a general relationship of identity, whereas $\mathit{identical}(a, a)$ concerns the specific fact of some constant a being identical to itself.

Definition 1 *An* adaptable grammar *is a quintuple* $\langle \Sigma, N, \Pi, [\![-]\!], R\rangle$ *where* Σ *is a finite* alphabet*,* N *a set of* nonterminal *symbols which are logical predicate symbols of arity at least 1,* Π *is a logic program,* $[\![-]\!]$ *is the* denotation function *which is a (partial) function from ground terms to grammars, and* R *is a set of grammar rules (below). Nonterminals and atoms defined as predicates in* Π *are assumed to be disjoint. Each nonterminal symbol has a distinguished argument called its* grammar argument *(or* grammar attribute*). A* grammar rule *is of the form*

$$\mathit{lhs}\ \texttt{-->}\ \mathit{rhs}.$$

where lhs is a nonterminal and rhs a finite sequence of elements which may be terminal or nonterminal symbols or first-order atoms defined by Π.

It may be noticed that the use of an explicit denotation function means that we avoid potential circularity problems, so that the definition can rely on standard sets of first-order terms. Intuitively, the denotation function defines a *ground representation* of grammars, i.e., the variables in rules are represented by constants or other ground objects. Without this, the application instances of rules (defined below) may include structures containing a copy of themselves, which requires a far more complicated mathematical foundation; see, e.g., [12] and its references to the mathematical background.

We apply a notation inspired by definite clause grammars [20] so that terminals in grammar rules are written in square brackets, and atoms referring the program component in curly brackets. Furthermore, the grammar argument of a nonterminal symbol is assumed to be the last of its arguments, and we drag it outside the standard parentheses and attach it by a hyphen, e.g., instead of `n(x,y,G)`, we prefer `n(x,y)-G` when `n` is a nonterminal of arity 3. In the text, we use italic letters for metavariables in the text, e.g., as in "let G be ..."; logical variables within grammar rules are indicated by capital letters in typewriter font (as `G` above), and constant symbols that denote such variables by single quotes, e.g., `'G'`. Our Prolog-based implementation uses a slightly different notation that will be explained.

Terms within rules that represent grammars are generally written using Prolog's list syntax but may be written in other ways depending on the actual denotation function. In order to simplify the notation, we expect that the denotation function only changes the rule sets, so that the other components are fixed within the derivations made from a given initial grammar. In the examples below, we assume Σ and N to be introduced implicitly by usage, and the logic program Π implicitly supplies any standard Prolog predicate that we may need.

However, it should be noticed that our definition allows any component to be modified during a discourse, including the 'knowledge base' Π and even the denotation function if needed; our implementation can be extended accordingly.

As it follows from the results of [10], the inclusion of logic programs in the formalism does not add anything new as they can be seen as grammars that generate the empty string. They are included for notational convenience only.

Example 1 We show here a very simple and technical example of an adaptable grammar in order to illustrate the details of the definition; more interesting examples will be given later. Let

$$G_0 = \langle \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}, \{\mathtt{n}/1\}, \emptyset, [\![-]\!]_0, \{r_1, r_2\}\rangle$$

be an adaptable grammar where the rules r_1 and r_2 are as follows.

r_1 : `n-G --> [a], n-modify(G)`

r_2 : `n-G --> [b]`

It is assumed, for this example, that only the rule set varies during syntactic derivations (to be defined). We can thus define the denotation function $[\![-]\!]_0$ as a function from lists of "rule terms" into grammars whose first components are fixed as those of G_0. We extend $[\![-]\!]_0$ so that it also applies to the representation of single clauses: any term that resembles a rule is mapped into a similar rule with the special quoted constants replaced by variables; for example, we have that $[\![\mathtt{n\text{-}'G'\text{-}\text{-}>[b]}]\!]_0 = r_2$. Any list of terms is mapped into the set of rules denoted by each of these terms, and finally to interpret the `modify` operator, we define $[\![\mathtt{modify}(gt)]\!]_0 = [\![gt]\!]_0 \setminus \{r_2\} \cup \{r_3\}$ where $r_3 = $ `n-G-->[c]`.

For reference in the following example, let t_1, t_2, t_3 be terms such with $[\![t_i]\!]_0 = r_i$, $i = 1, 2, 3$.

Definition 2 (derivation) *Given an adaptable grammar $\langle \Sigma, N, \Pi, [\![-]\!], R\rangle$, an* application instance *of one of its rules $r \in R$ is a ground instance r' of r in which grammar arguments denote grammars and any logical atom in r' is satisfied in Π. Whenever $\alpha\beta\gamma$ is a sequence of ground grammar symbols, β being a nonterminal of the form N-G with an application instance β `-->` δ of a rule in $[\![G]\!]$, we write*

$$\alpha\beta\gamma \Rightarrow \alpha\delta'\gamma$$

where δ' is a copy of δ with any atom referring to Π taken out. The relation $\Rightarrow^*$ refers to the reflexive, transitive closure of $\Rightarrow$.

The language *defined by a given ground nonterminal N-G is the set of terminal strings σ for which N-$G \Rightarrow^* \sigma$.*

The application instances of a given grammar can be thought of as an infinite set of context free rules. When a particular nonterminal $N = n\text{-}\langle \textit{grammar-term}\rangle$ is rewritten, only those application instances whose left-hand side coincides with N can be used. In many cases, this amounts to a finite set, which may intuitively be understood as the context-free potential for a particular node in a syntax tree.

Syntax trees are written in the usual way, but extending each nonterminal node with the available grammar or rule set that is available for the expansion of that node. When no confusion occurs, we may interchange grammars and the terms that denotes them and even write $N - G$ as a node when N refers to a nonterminal with its grammar attribute (that denotes G).

Example 2 We continue example 1 above. We notice that

$$N_0 = \texttt{n-[}t_1\texttt{,}t_2\texttt{]}$$

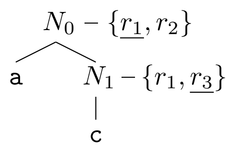
is a nonterminal whose grammar part denotes the grammar G_0; G_0 in turn has an application instance

$$\texttt{n-[}t_1\texttt{,}t_2\texttt{] --> [a], } N_1$$

where $N_1 = \texttt{n-modify([}t_1\texttt{,}t_2\texttt{])}$. We have thus the derivation steps

$$N_0 \Rightarrow \texttt{a}N_1 \Rightarrow \texttt{ac},$$

where the last step is due to the facts that N_1's grammar part denotes a grammar $\{r1, r3\}$ and that an application instance of r_3 can apply to rewrite N_1 into terminal symbol `c`. We can illustrate this by a syntax tree in which each node is decorated with the actual rules that can apply (more precisely, whose application instances can apply), with an underlining of the chosen one.


$N_0 - \{\underline{r_1}, r_2\}$
a
$N_1 - \{r_1, \underline{r_3}\}$
c

We notice also that $N_0 \Rightarrow^* \texttt{b}$, but not $N_0 \Rightarrow^* \texttt{ab}$.

3 Plain Vanilla Implementation of Adaptable Grammars

Adaptable grammars as we have defined them above can be implemented in Prolog in a straightforward way, based on well-tested techniques. We stress that the code shown here can be executed directly as it is in any standard Prolog system; no additional facilities or hidden devices are needed.

We will assume a non-ground representation of grammars, i.e., using logical variables to denote logical variables. This representation has the advantage that programs can be written very succinctly but requires a bit of care to avoid mixing up quantification levels. We take this for granted and refer to standard literature on this topic, e.g., [12], and avoid these potential technical problems in our examples.

Instead of magically predicting the right application instances of grammar rules in order to analyze a given string, we use unification when a rule is applied, and embedded code chunks are executed in the order they are encountered in the current derivation; these are also standard techniques for execution of Prolog programs and language recognition with Definite Clause Grammars [20], so we take correctness for granted. Our implementation is similar to a meta-interpreter implementation of Definite Clause Grammars with the single extension that grammar rules are picked in the grammar argument instead of in Prolog's global database. An extra predicate for renaming variables in grammar rules is needed, implemented here in a traditional one-line fashion. The following few code lines are sufficient to check whether a given string S can be derived from an adaptable grammar nonterminal NG by the call `derive(`NG`,` S`)`.

```
derive(NG, S):- derive(NG, S, []).

derive([], S, S).

derive([T|Ts], [T|S1], S2):- derive(Ts, S1, S2).

derive({Code}, S, S):- call(Code).

derive((A,B), S1,S3):- derive(A, S1, S2), derive(B, S2, S3).

derive(N-G, S1, S2):-
    renameVars(G,GFresh),
    member( (N-G --> Rhs), GFresh),
    derive(Rhs,S1,S2).

renameVars(X,Y):- asserta(quax(X)), retract(quax(Y)).
```

Example 3 We continue examples 1 and 2 by showing the encoding of grammar G_0 in Prolog for demonstrating the use of the interpreter above. We need to modify the grammar a little, as the inherent denotation function does not understand the `modify` operator. Instead, we define a Prolog predicate with the same purpose.

```
modify([R1,R2],[R1,R3]):- R3=(n-G-->[c]).
```

It is convenient to store the grammar as a Prolog fact exemplified as follows.

```
gram( [ (n-G --> [a], {modify(G,G1)}, n-G1),
        (n-G --> [b]) ]).
```

Testing that the string `ab` is in the language is then done as follows.

```
| ?- gram(G), derive(n-G,[a,c]).
G = [...] ?
yes
```

The appendix demonstrates tests for the grammars shown in remaining part of the paper.

4 Important Non-Context-Free Languages in Adaptable Grammars

In formal linguistics, three different phenomena are often emphasized as central for natural language; they are called *reduplication*, *crossed dependencies* and *multiple agreements.* These phenomena are exposed in the following languages (see, e.g., [18]):

$$\begin{aligned} L_1 &= \{rcr \mid r \in \{a,b\}^*\} \\ L_2 &= \{a^n b^m c^n d^m \mid n,m \geq 1\} \\ L_3 &= \{a^n b^n c^n \mid n \geq 1\} \end{aligned}$$

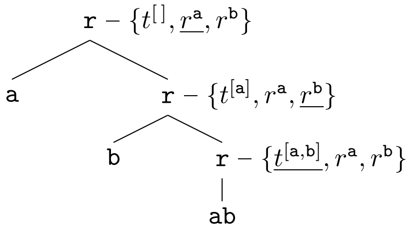
These are straightforward to describe in adaptable grammars as we will show below. We introduce the language

$$RR = \{rr \mid r \text{ is a string of } \mathtt{a}\text{s and } \mathtt{b}\text{s}\}.$$

as a prototype. We can characterize RR by a grammar G_{RR} that generates the first half part of the string in a standard way and in parallel maintains a rule, called the *terminator rule,* for generating the second half. Initially, when no `a`s and `b`s have been derived, the terminator rules generates the empty string. More precisely, G_{RR} consists of the following three rules, $t^{[\,]}$, $r^{\mathtt{a}}$ and $r^{\mathtt{b}}$.

$$\begin{aligned} t^{[\,]}: &\quad \texttt{r-G} \quad \texttt{-->} \quad \texttt{[]} \\ r^{[\mathtt{a}]}: &\quad \texttt{r-G} \quad \texttt{-->} \quad \texttt{[a]}, \{\textit{add-to-terminator}(\mathtt{a})\}, \texttt{r-G1} \\ r^{[\mathtt{b}]}: &\quad \texttt{r-G} \quad \texttt{-->} \quad \texttt{[b]}, \{\textit{add-to-terminator}(\mathtt{b})\}, \texttt{r-G1} \end{aligned}$$

where *add-to-terminator*(*letter*) is an action that changes the terminator rule of grammar given by `G` by adding *letter* in front of its right-hand side in order to obtain a revised grammar for `G1`. The following syntax tree, indicated in a notation similar to example 2, represents the derivation of the string '`abab`'; the notation $t^{[string]}$ indicates the terminator rule `r-G --> `[*string*].


$\texttt{r} - \{t^{[\,]}, \underline{r^{\texttt{a}}}, r^{\texttt{b}}\}$
a
$\texttt{r} - \{t^{[\texttt{a}]}, r^{\texttt{a}}, \underline{r^{\texttt{b}}}\}$
b
$\texttt{r} - \{\underline{t^{[\texttt{a,b}]}}, r^{\texttt{a}}, r^{\texttt{b}}\}$
ab

Changing the G_{RR} grammar above to a grammar for L_1 is a matter of changing the terminator rule of the initial grammar into `r-G-->[c]`. The principle applied in the grammar for RR provides immediately the clue for describing L_2 and L_3. For L_2 we can use an initial grammar consisting of the following rules (sketched).

```
s0-G  -->  [a], s1-G
s1-G  -->  [a], {code1}, s1-G1
s1-G  -->  [b], s2-G
s2-G  -->  [b], {code2}, s2-G1
s2-G  -->  Cs-G, Ds-G
Cs-G  -->  [c]
Ds-G  -->  [d]
```

The embedded code fragment *code*$_1$ adds a `c` to the right-hand side of the rule for `Cs`; thus one application of the first rule and $n-1$ application of the second create the rule `Cs-G --> [c]`^{n}. Similarly, *code*$_2$ adds a `d` to the rule for `Ds` in order to form the rule that produces `d`^{m} in the final derivation step.

We have chosen the language RR instead of L_1, as it may be more difficult to model in some formalisms due to the lack of a "middle demarcation" c. Changing our grammar above to L_1 is a matter of changing the first rule of the initial grammar into `r-G --> [c]`. The language L_3 can be described in exactly the same way, so that each time the rule for generating an `a` is applied, rules for `b`s and `c`s are extended with one more letter. We have shown grammars for L_2 and L_3 in the appendix, that work in a slightly different and even simpler way.

5 Adaptable Grammars for Grammar-Definitions and other Meta-Languages

The language in a discourse may develop as the discourse goes along, new usages are introduced or usages may achieve new meanings, which can be "stored" in grammar rules. We may say that the semantic-pragmatic context (which is something very different from the syntactic context considered in Contextual Grammars [17]) is growing along with the discourse and in this way enriches the linguistic potential.

We consider here meta-level architectures, as they arise in parser generators (e.g., YACC [14]), as archetypical in that they provide explicit language constructs for linguistic enrichment.

Indeed, one of the most striking features of adaptable grammars is their ability to characterize meta-level architectures. A formal grammar, for example, may itself by written in an idiosyncratic language of the sort typically called a meta-language. Context-free grammars may be written with the left and right-hand sides separated by, e.g, "`::=`" or "`-->`".

We describe here the syntax of a meta-level system that accepts the definition of a context-free grammar in a specific syntax together with an arbitrary sample string, supposed to be described by that context-free grammar. The following is an example of a correct formulation in this meta-level system.

```
start ::= n1 n2 .
n1 ::= a b .
n2 ::= b a .
sample a b b a
```

For simplicity of the grammar, the nonterminals of the novel grammar must be chosen among "`start`", "`n1`", and "`n2`", and terminals must be chosen among "`a`" and "`b`"; "`start`" is assumed always to be the start nonterminal.

The meta-level grammar for this system can be written as the following adaptable grammar that is referred to as G_0. For the denotation function, we assume that a grammar is denoted as a list of ground terms, each representing a particular rule. We use below, the logic programming notation for list construction such that when L refers to a list, E to e new list element, then `[`E`|`L`]` refers to a new list starting with E and followed by elements of L. Comma structures are in used a similar way for constructing rule bodies. The nonterminal '`e`' (deriving the empty string) will be included at the end of each generated rule for simplicity only; it can easily be suppressed by adding a bit of extra code.

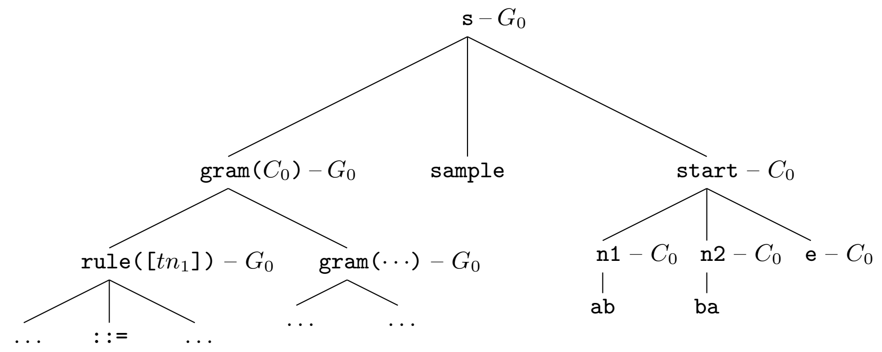
```
                      s-G  -->  gram(C)-G, [sample], start-C
   gram([(e-‘G’-->[])])-G  -->  []
           gram([R|Rs])-G  -->  rule(R)-G, gram(Rs)-G
         rule((H-->B))-G  -->  head(‘G’,H)-G,
                                  [::=], body(‘G’,B)-G
            head(‘G’,H)-G  -->  nonterm(‘G’,H)-G
    body(‘G’, ([T],Bs))-G  -->  term(T)-G, body(‘G’,Bs)-G
       body(‘G’,(N,Bs))-G  -->  nonterm(‘G’,N)-G,
                                  body(‘G’,Bs)-G
        body(‘G’,e-‘G’)-G  -->  [.]
```

```
                term(a)-G  -->  [a]
                term(b)-G  -->  [b]
nonterm('G',start-'G')-G  -->  [start]
   nonterm('G',n1-'G')-G  -->  [n1]
   nonterm('G',n2-'G')-G  -->  [n2]
```

When the first rule is applied to s-t_{G_0}, where t_{G_0} is a term denoting G_0, firstly gram(C)-G_0 is processed, producing a new adaptable grammar C_0 whose term representation t_{C_0} is bound to the variable C; then start-t_{C_0} is processed for the investigation of a sample sentence. In the example above, C_0 will consist of the following rules.

```
start-G  -->  n1-G, n2-G, e-G
   n1-G  -->  [a], [b], e-G
   n2-G  -->  [b], [a], e-G
    e-G  -->  []
```

We can illustrate the actual derivation in terms of the following syntax tree; the notation tn_1 refers to a term that denotes the second clause, i.e., the one for n1.



Programming languages include also devices that may be seen as language extenders. Declaring, say, a variable n of type int means that we can now write n in positions of the program where an int is expected; before the declaration was added, such usages were illegal, ungrammatical so to speak. In other words, the indicated variable declaration can be seen as the creation a new grammar rule int-G --> [n]. Programming languages were in focus in earlier publications on adaptable grammars; see [3, 4, 5] for more examples of programming language constructs described in this way but with a different notation. Books of math are another kind of texts in which new nomenclature is introduced in an explicit way and for which adaptable grammars may be used.

6 Conclusion

We have defined a highly expressive grammar formalism which captures non-context-free language features by treating the grammar itself as a dynamic entity, which can be elaborated and changed by the application of grammar rules. We have shown a plain vanilla meta-interpreter in Prolog which makes it possible to test and experiment with adaptable grammars. We have justified the usefulness of this grammar formalism by showing grammars for important examples of non-context-free languages; we notice that these grammars were rather straightforward and we needed to take the step to a more complex meta-level system in order to illustrate the formalism's power for a more detailed analysis of the text and synthesis of new rules.

A more powerful and logically more satisfactory (but, alas, less transparent) implementation may be possible using a constraint-based denotation function, which may be implemented using 'instance of' constraints [6] with a potential for producing new rules in abductive or even inductive ways from rule applications. Constraint-based abduction in discourse analysis, as applied e.g., by [2, 8], may also be interesting to combine with the approach described in the present paper.

We have presented an approach to context-sensitivity that allows for creating rules for new expressions that come into use and to remove or change rules for usages that go out of fashion (or perhaps become idiomatic expressions whose sub-expressions loose their meaning). We hope in this way to provide inspiration for future research in linguistics that concerns the evolution of language, whether this be within a discourse or in a broader perspective, maybe over generations of language users or even in the evolution of biological species.

It may be possible that other frameworks than the DCG grammars used here may show to be more appropriate to extend with adaptable capabilities for this sort of research. With the present work, we intended to emphasize and expose the principle of adaptability in linguistic specifications and to provide an initial light-weight implementation framework suitable for further experimentation.

A Example grammars in full details

Here we show grammars for important languages mentioned in section 4; they are defined in the format of Prolog source text, so that they can be tested in any standard Prolog system using the interpreter shown in section 3 above. We emphasize again that the Prolog implementation can be used directly for the reader's experiments, no hidden auxiliaries are used.

The grammar for $L_{RR} = \{rr \mid r \text{ is a string of } \texttt{a}\text{s and } \texttt{b}\text{s}\}$ can be represented in Prolog by the following clause.

```
rrgram(  [ ( r-G --> [] ),
           ( r-G --> [a],{C1}, r-G1 ),
           ( r-G --> [b],{C2}, r-G1 )    ]):-

 C1 = (G =[(S-->List)|RestG],
       append(List,[a],NewList),
       G1=[(S-->NewList)|RestG]),
 C2 = (G =[(S-->List)|RestG],
       append(List,[b],NewList),
       G1=[(S-->NewList)|RestG]).
```

It could have been written equivalently as a Prolog fact, but we use the substitutions in the rule body in order to obtain a more readable presentation; this way, variables `C1` and `C2` abbreviate the embedded code that transforms the grammar attribute. The grammar can be tested as follows.

```
?- rrgram(G), derive(r-G, [a,b,b,a,a,b,b,a]).
G = [...] ?
yes
```

The language $L_2 = \{a^n b^m c^n d^m \mid n, m \geq 1\}$, which is intended to represent the essence of crossed dependencies, can be defined by a grammar as follows. As also done above, we represent the grammar as a Prolog rule, so that we can use unifications in the body for making complex expressions more readable; notice the introduction of an auxiliary predicate that characterize the language.

```
g2gram(
  [ ( s-G --> as(Cs)-G, bs(Ds)-G, cs-[Rc], ds-[Rd]  ),
    ( as([c])-G --> [a] ),
    ( as([c|Cs])-G --> [a], as(Cs)-G ),
    ( bs([d])-G --> [b] ),
    ( bs([d|Ds])-G --> [b], bs(Ds)-G )
   ]):-

   Rc = (cs-G1 --> Cs),
   Rd = (ds-G1 --> Ds).

g2(S):- g2gram(G), derive(s-G, S).
```

The grammar can be tested as follows.

```
| ?- g2([a,a,b,b,b,c,c,d,d,d]).
yes
| ?- g2([a,a,b,b,b,c,c,d,d,d,d]).
no
```

The language $L_3 = \{a^n b^n c^n \mid n \geq 1\}$ which is intended to represent the essence of multiple agreement, can be defined by a grammar as follows.

```
g3gram(
  [ ( s-G --> as(Bs,Cs)-G, bs-[Rb], cs-[Rc] ),
    ( as([b],[c])-G --> [a] ),
    ( as([b|Bs],[c|Cs])-G --> [a], as(Bs,Cs)-G )
```

```
   ]):-

   Rb = (bs-G1 --> Bs),
   Rc = (cs-G1 --> Cs).

g3(S):- g3gram(G), derive(s-G, S).
```

The grammar can be tested as follows.

```
| ?- g3([a,a,b,b,c,c]).
yes
| ?- g3([a,a,b,c,c]).
no
```

We will mention a simple but useful technique which is very useful for the reader who wishes to develop and test his or her own, new adaptable grammars using our interpreter. Typically the first version of a grammar contains bugs that the developer needs to locate and correct. In principle, Prolog's tracer can be used, but (here as in general) experience shows that it produces far too much information. We suggest instead to use test prints that can written directly into the grammar as embedded code. Consider as an example, the first rule of the grammar for L_2 above. Modifying it into the following does not change its logical meaning, but it will print the rules created for the recognition of the `c` and `d` subsequence.

```
l2gram(
  [ ( s-G --> as(Cs)-G, {write(Rc)}, bs(Ds)-G,
              {write(Rc)}, cs-[Rc], ds-[Rd]  ),
    ...
   ]):-
   Rc = (cs-G1 --> Cs),
   Rd = (ds-G1 --> Ds).
```

Finally, the grammar for the meta-level system shown in section 5 can be used almost directly by the Prolog implementation. The only modification needed to the text of the grammar is to replace quoted variable names such as `'G'` by new variables, e.g., `Gnew`, not used as proper variables in the "top-level" grammar.

Acknowledgment

This work is supported by the project "Logic-statistic modelling and analysis of biological sequence data" funded by the NABIIT program of the Danish Strategic Research Council.

References

[1] Boris Burshteyn. On the modification of the formal grammar at parse time. *SIGPLAN Notices*, 25(5):117–123, 1990.

[2] H. Christiansen and V. Dahl. Meaning in Context. In Anind Dey, Boicho Kokinov, David Leake, and Roy Turner, editors, *Proceedings of Fifth International and Interdisciplinary Conference on Modeling and Using Context (CONTEXT-05)*, volume 3554 of *Lecture Notes in Artificial Intelligence*, pages 97–111, 2005.

[3] Henning Christiansen. Syntax, semantics, and implementation strategies for programming languages with powerful abstraction mechanisms. In *18th Hawaii International Conference on System Sciences*, volume 2, pages 57–66, 1985.

[4] Henning Christiansen. The syntax and semantics of extensible languages. *Datalogiske skrifter* 14 (Tech. rep). Computer Science Section, Roskilde University, Denmark, 1988.

[5] Henning Christiansen. A survey of adaptable grammars. *SIGPLAN Notices*, 25(11):35–44, 1990.

[6] Henning Christiansen. Automated reasoning with a constraint-based metainterpreter. *Journal of Logic Programming*, 37(1-3):213–254, 1998.

[7] Henning Christiansen. Adaptable grammars for non-context-free languages. In Joan Cabestany, Francisco Sandoval, Alberto Prieto, and Juan M. Corchado, editors, *IWANN (1)*, volume 5517 of *Lecture Notes in Computer Science*, pages 488–495. Springer, 2009.

[8] Henning Christiansen and Verónica Dahl. HYPROLOG: A new logic programming language with assumptions and abduction. In Maurizio Gabbrielli and Gopal Gupta, editors, *ICLP*, volume 3668 of *Lecture Notes in Computer Science*, pages 159–173. Springer, 2005.

[9] Marina de la Cruz Echeandía and Alfonso Ortega de la Puente. A christiansen grammar for universal splicing systems. In José Mira Mira, José Manuel Ferrández, José R. Álvarez, Félix de la Paz, and F. Javier Toledo, editors, *IWINAC (1)*, volume 5601 of *Lecture Notes in Computer Science*, pages 336–345. Springer, 2009.

[10] Pierre Deransart and Jan Maluszynski. Relating logic programs and attribute grammars. *Journal of Logic Programming*, 2(2):119–155, 1985.

[11] Alfonso Caracciolo di Forino. Some remarks on the syntax of symbolic programming languages. *Communications of the ACM*, 6(8):456–460, 1963.

[12] P. M. Hill and J. Gallagher. Meta-programming in logic programming. In *Handbook of Logic in Artificial Intelligence and Logic Programming*, pages 421–497. Oxford Science Publications, Oxford University Press, 1994.

[13] Quinn Tyler Jackson. *Adapting to Babel: Adaptivity and Context-Sensitivity in Parsing.* Ibis Publications, Plymouth, Massachusetts, USA, 2006.

[14] Stephen C. Johnson. Yacc: Yet another compiler-compiler. Technical Report CSTR 32, Bell Laboratories, Murray Hill, N.J., USA, 1975.

[15] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[16] John W. Lloyd. *Foundations of logic programming, 2nd, extended ed.* Springer-Verlag, 1987.

[17] Solomon Marcus. Contextual grammars. *Rev. roum. de math. pures et appl*, 14:1473–1482, 1969.

[18] Solomon Marcus, Gheorghe Paun, and Carlos Martín-Vide. Contextual grammars as generative models of natural languages. *Computational Linguistics*, 24(2):245–274, 1998.

[19] Alfonso Ortega, Marina de la Cruz, and Manuel Alfonseca. Christiansen grammar evolution: Grammatical evolution with semantics. *IEEE Trans. Evolutionary Computation*, 11(1):77–90, 2007.

[20] Fernando C. N. Pereira and David H. D. 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.

[21] John N. Shutt. Recursive adaptable grammars. Master's thesis, Computer Science Department, Worcester Polytechnic Institute, Worcester Massachusetts, USA, 1993.

[22] Adriaan van Wijngaarden, B. J. Mailloux, J. E. L. Peck, Cornelis H. A. Koster, Michel Sintzoff, C. H. Lindsey, Lambert G. L. T. Meertens, and R. G. Fisker. Revised report on the algorithmic language algol 68. *Acta Informatica*, 5:1–236, 1975.

[23] Ben Wegbreit. *Extensible programming languages.* Harward University, Cambridge, Massachusetts, USA, 2006. (Garland Publishing Inc., New York & London, 1980).

[24] Thomas Weise. Global optimization algorithms - theory and application. Electronic manuscript (e-book); linked checked January 2008, 2007.