# Constraint Solving and Language Processing

2nd International Workshop, CSLP 2005
Sitges, Spain, 5 October 2005

Proceedings

Edited by

Henning Christiansen
Jørgen Villadsen

# Preface

This volume contains the proceedings of CSLP 2005, the Second International Workshop on Constraint Solving and Language Processing which takes place in Sitges, Spain, on 5 October, 2005. CSLP 2005 is affiliated with ICLP'05, 21st International Conference on Logic Programming, also co-located with CP'05, 11th International Conference on Principles and Practice of Constraint Programming. From a historical perspective and for attracting attention among interested researchers, this forum is perfect for CSLP, and we would like to thank the organizers of ICLP, especially its workshop chair Hai-Feng Guo, for hosting the workshop.

We are honoured to present our invited speaker, Jerry R. Hobbs, whose research has been an influent and important source of inspiration for all other researchers in the field for around three decades.

Constraint Solving (CS), in particular Constraint Logic Programming (CLP), is a promising platform, perhaps the most promising present platform, for bringing forward the state of the art in language processing. The data subjected to processing via constraint solving may include written and spoken language, formal and semiformal language, and even general input data to multimodal and pervasive systems.

CLP and CS have been applied in projects for shallow and deep analysis and generation of language, and to different sorts of languages. The view of grammar expressed as a set of conditions simultaneously constraining and thus defining the set of possible utterances has influenced formal linguistic theory for more than a decade. CLP and CS provide flexibility of expression and potential for interleaving the different phases of language processing, including handling of pragmatic and semantic information.

This volume contains papers accepted for the workshop based on an open call, and each paper has been reviewed by two or three members of the program committee. As editors, we would also like to thank the other members of the organization committee, Philippe Blache, Veronica Dahl, and Gerald Penn, whose involvement has been important for the establishment of the forum around the CSLP workshops. We hope that the present volume together with the collection of revised papers from the 1st CSLP, published as Lecture Notes in Artificial Intelligence, vol. 3438, can provide inspire future research in this promising field.

We want to thank the program committee listed below, the invited speaker, and all researchers who submitted papers to the workshop and all participants in the CSLP workshops 2004 and 2005. The workshop is supported by the CONTROL project, CONstraint based Tools for RObust Language processing, funded by the Danish Natural Science Research Council, and Computer Science Section at Roskilde University, Denmark.

Roskilde, September 2005

Henning Christiansen
Jørgen Villadsen

## Organizing Committee

Philippe Blache
Henning Christiansen
Veronica Dahl
Gerald Penn

## Program Committee

Philippe Blache, Aix-en-Provence, France
Henning Christiansen, Roskilde, Denmark (Chair)
Veronica Dahl, Simon Fraser University, Canada
Denys Duchier, Lille, France
John Gallagher, Roskilde, Denmark
Claire Gardent, LORIA, France
Michael Johnston, ATT, USA
Shalom Lappin, London, UK
Bernd Meyer, Monash, Australia
Jørgen Fischer Nilsson, Technical University of Denmark
Gerald Penn, Toronto, Canada
Kiril Simov, Bulgarian Academy of Science
Peter Skadhauge, Copenhagen Business School, Denmark
Jørgen Villadsen, Roskilde, Denmark

# Contents

# Syntax, Metonymy, and Intention

Jerry R. Hobbs

Information Sciences Institute
University of Southern California
Marina del Rey, California
USA

**Abstract.** Metonymy is referring to one entity by describing a functionally related entity. When this is formalized in an "Interpretation as Abduction" framework, we see that it is a very powerful mechanism for solving a number of problems that have hitherto been viewed as syntactic. In these cases, the coercion function that mediates the metonymy comes from material that is explicit in the sentence. For example, in "John smokes an occasional cigarette", it is the smoking, rather than the explicit argument of "occasional" (the cigarette), that is occasional. There is a coercion from the cigarette to the smoking event, where the coercion relation is provided by the "smokes" predication itself. Other phenomena analyzed in this manner are extraposed modifiers, container nouns, the collective-distributive ambiguity for some plural noun phrases, small clauses in disguise such as "This country needs literate citizens", the assertion of grammatically subordinated material, and monotone decreasing quantifiers.

It will also be shown how material in lexical decompositions can be used as coercion relations to make sense of a sentence like "John opened the door again" where he was not the person who opened it in the first place.

Then the intentional and cognitive "wrappers" around the content of a sentence will be introduced; essentially, an explanation of the occurrence of an utterance of a string of words $w$ by a speaker $i$ to a hearer $u$ is that $i$ has the goal of $u$ "cognizing" the eventuality $e$, where $w$ is a string of words that conveys or describes $e$. Then it is shown how epistemic and speech act conditionals and metalinguistic negation can be viewed as examples of metonymy, where the "goal" and "cognize" predicates are used as the coercion functions.

It will also be shown how explicit representation of implication relations used in interpreting metaphors can be used as coercion relations to shield metaphorical statements against being false. For example, "'John is an elephant' describes John's being large" can be coerced into "'John is large' describes John's being large", using the implication relation between "elephant" and "large" as the coercion relation, so that what is asserted by "John is an elephant" is precisely that John is large – presumably a true statement.

A common, uniform framework for syntax, semantics, and pragmatics, where knowledge and the content of texts are expressed in first-order logic, in which abduction is used as the inference mechanism, and in

which interpretation is viewed as global optimization and constraint satisfaction, facilitates the investigation of phenomena that lie in the frontier regions where syntax, semantics and pragmatics meet.

# Extracting Selected Phrases through Constraint Satisfaction

Veronica Dahl[1] and Philippe Blache[2]

[1] Simon Fraser University
Vancouver, Canada
veronica@cs.sfu.ca
[2] LPL, Université de Provence
Aix-en-Provence, France
pb@lpl.univ-aix.fr

**Abstract.** We present in this paper a CHR based parsing methodology for parsing *Property Grammars*. This approach constitutes a flexible parsing technology in which the notions of derivation and hierarchy give way to the more flexible notion of constraint satisfaction between categories. It becomes then possible to describe the syntactic characteristics of a category in terms of satisfied and violated constraints.
Different applications can take advantage of such flexibility, in particular in the case where information comes from part of the input and requires the identification of selected phrases such as *NP, PP*, etc. Our method presents two main advantages: first, there is no need to build an entire syntactic structure, only the selected phrases can be extracted. Moreover, such extraction can be done even from incomplete or erroneous text: indication of possible kinds of error or incompleteness can be given together with the proposed analysis for the phrases being sought.

## 1 Introduction

Extracting selected phrases from written or spoken text is an important step for many useful applications of language processing. For instance, concept extraction or text summarization can benefit from a preliminary identification of all noun phrases or verbs, to be further processed, e.g. through consulting a concept hierarchy; question answering can focus on noun phrases or verbs at least for the replies to be given; command oriented systems might focus on verb phrases; temporal systems, on time adverbial phrases, etc.

Ideally we want to be able to extract concepts from text produced in real life conversation, which typically is incomplete, often not perfectly grammatical, and sometimes erroneous. Imperfections can result from normal human error in actual speech, or be introduced by machines, as in the case of text produced from speech recognition systems, which are renowned for their error-proneness.

Flexible parsing techniques offer great advantages in this perspective. Among possible solutions, constraint-based approaches allow us to use the same parsing mechanism (constraint satisfaction) whether the grammar is incomplete, heterogeneous, etc. Moreover, provided that no extra mechanisms than satisfaction are used, constraints can be relaxed in accordance to user-defined criteria. In particular, this can be done selectively (for some sentences but not others). For instance, we may want to express that a noun requires a determiner inside a noun phrase, unless we are dealing with a generic statement, as in "Lions sleep". To this effect, we can test the conditions relevant to genericity in the body of the rule that relaxes the constraint imposing a determiner, so that only upon them being satisfied will the constraint be relaxed.

Property-based linguistic models (see [Bès99a,Bès99b], [Blache05]) view linguistic constraints as properties between sets of categories, rather than in the more traditional terms of properties on hierarchical representations of completely parsed sentences. This view has several advantages, such as allowing for mistakes to be detected and pointed out rather than blocking the analysis altogether, and facilitates dynamic processing of text produced on the fly, as needed for the growing number of applications involving speech.

In this paper we refine a methodology for *Property Grammars* (cf. [Blache05]) first proposed in [Dahl04b] which relies exclusively on constraints. It controls the parse through head-driven analysis, provides a direct interpretation of this formalism while preserving all its theoretical properties at the implementation level, and can focus on specific phrases- in the case of our toy implementation, noun phrases- as mandated by a user's modular and easy to change command. As the implementation language, we use a specialized grammatical formalism called CHRG (Constraint Handling Rule Grammars) described in [Christiansen01] on top of CHR [Frühwirth98].

## 2 Representing syntax with constraints

The basic idea of *Property Grammars* is to represent different kinds of syntactic information separately. In this approach, syntactic structure is not expressed in terms of hierarchy, but only by means of relations between categories. We describe in this section how to represent such relations in terms of constraints and take advantage of this in the perspective of a direct constraint-based implementation.

### 2.1 Background: Property Grammars

In our approach, syntactic properties rely on relations that do not have specific topological constraints (they can for example be crossed). Categories are described by means of such relations. As a consequence, the notion of constituency is no longer crucial for the description process: a category is specified by a set of properties rather than by a set of constituents. The fact that several categories belong to a network of relations indicates that they characterize an upper-level

category. A syntactic category is then described by a set of properties which represent relations between other categories (lexical or syntactic).

In this approach, the goal is to make explicit all the different relations that can exist. We distinguish in this perspective the following types of information:

- linear precedence, which is an order relation between categories,
- subcategorization, which indicates coocurrence relations between categories or sets of categories,
- the impossibility of cooccurrence between categories,
- the impossibility for a category to be repeated,
- the minimal set of obligatory constituents (usually one single constituent) which is the head,
- semantic relations between categories, in terms of dependency.

These different kinds of information correspond to different properties, respectively: *linearity, requirement, exclusion, unicity, obligation, dependency.* Such information can always be expressed in terms of relations between categories, as shown in the following examples:

- Linear precedence: $Det \prec N$ (a determiner precedes the noun)
- Dependency: $AP \rightsquigarrow N$ (an adjectival phrase depends on the noun)
- Requirement: $V[inf] \Rightarrow to$ (an infinitive comes with *to*)
- Exclusion: $seems \not\Rightarrow ThatClause[subj]$ (the verb *seems* cannot have *That* clause subjects)

All syntactic categories are characterized by a set of relations that forms a connected graph. The syntactic description of a language consists of all the different relations that can be expressed between categories. A relation (also called a property) can be conceived as a constraint on the set of categories. A grammar is then a set of constraints and satisfiability becomes the core of the parsing process (see [Blache01]). What is interesting in this approach is that no implicit information, for example under the form of a specific mechanism, is needed. In particular, there is no need to build a structure before being able to verify its properties as it is the case with classical generative approaches (although for convenience given what researchers are used to, we do provide a tree as well as a side effect of parsing). Moreover, using satisfiability alone has important consequences in the conception of the syntactic structure. Evaluating a constraint system for a given set of categories allows us to specify precisely the set of properties that are verified. In the same way, in the case of ill-formed input, such evaluation identifies precisely the set of satisfied and violated constraints. Such a result is then of deep interest in the sense that it identifies precisely all the specificities of an input. In *Property Grammars* this information constitutes the output of a parse, which is but the status of the constraint system after evaluation.

### 2.2 The parsing schema

The basic mechanism in constraint satisfaction problems is to find, for a given set of variables, an assignment that satisfies the constraint system. In the problem addressed here, the variables are taken from the set of categories. An assignment is given from an input (i.e. the sentence to be parsed). Starting from the set of lexical categories corresponding to the words of the sentence, all possible assignments (i.e. subsets of categories) are evaluated. When a syntactic category is characterized, it is added to the set of categories to be evaluated. This approach is basically incremental in the sense that any subset of categories can be evaluated. This means that an assignment $A$ can be completed by adding other categories. When syntactic categories are inferred after the first step of the process, it is then possible to complete the first assignments (made with lexical categories) with new syntactic ones.

We have seen that in Property Grammars, a category is described by a set of constraints. But reciprocally, it is possible to identify a category from a given property. This is typically the case with properties expressing relationships between categories, such as linearity, requirement, obligation and dependency. Evaluating such properties makes it possible to infer that the syntactic category to which the property is attached is being characterized. We have referred to these properties under the collective name of *selection* constraints.

The role of selection constraints is central to our approach. The reason such constraints allow us to select the characterized category is that they are local to this category. Moreover, in some cases they have a global scope over the category : their satisfiability value (i.e. satisfied or violated) cannot change for a given category whatever the subset of constituents. As soon as the constraint can be evaluated, this value is permanent. For example, when a linearity or a dependency constraint is satisfied, adding new constituents to the category cannot change this fact. Other kinds of constraints have to be re-evaluated at each stage. For instance, when adding a new category, we need to verify that unicity and exclusion are still satisfied. In all that follows, we call the latter *filtering constraints*. Contrary to *selection constraints*, one cannot infer the materialization of a syntactic category from their evaluation. They play a filtering role in the sense that they rule out some construction. Yet another type of constraint, which we call *recoverable constraints* can succeed by the incorporation of one more category into a given phrase for which, without this added category, the constraint failed.

Let us examine what the consequences are on constraint evaluation. As explained above, the principle consists in completing original assignments with new categories when they are inferred. Insofar as the evaluation of selection constraints (as soon as this evaluation can be performed) is valid through a complete assignment, whatever its constituents, it is not necessary to re-calculate it. In other words, when an assignment $A$ is made by completing another assignment $A'$, the set of selection constraints of $A'$ is inherited by $A$. We describe in the next section the different types of properties and their consequence for new assign-

ments in more detail, with respect to a specific instance of the general parsing schema we present here.

In the following, we note selection and filtering constraints as $R_{select}(C, XP)$ and $R_{filter}(C, XP)$ in which $C$ is the constraint and $XP$ the syntactic category to which the constraint is associated. For some assignment $A$, a constraint is relevant (or can be evaluated) when the categories of $A$ are a subset of the categories involved in $C$. We note the fact that $A$ can be evaluated for some constraint as follows: $A/R_{select}(C, XP)$. We note the set of filtering and selection constraints for a given category $XP$ by $R(C, XP) = R_{select}(C, XP) \cup R_{filter}(C, XP)$. Finally, we note the state of the constraint system $\Sigma$ for an assignment $A$ after evaluation by $SAT(A, \Sigma)$. Each category is indexed by its boundaries, noted $c_{(i,j)}$.

Let $K$ be the set of categories, noted $c_i$, let $i, j, k$ some indexes such that $i < j < k$:

1. $A \leftarrow \{c_i, ..., c_j\}$
2. if $A/R_{select}(C, XP)$
3. instantiate $XP_{(i,j)}$
4. $\Sigma = \bigcup R(C, XP)$
5. $Char(A) \leftarrow SAT(A, \Sigma)$
6. while $SAT(A, \Sigma)$ acceptable
7. $A \leftarrow A \cup \{c_k\}$
8. $\Sigma = \bigcup R(C, XP)$
9. $Char(A) \leftarrow SAT(A, \Sigma)$
10. $k \leftarrow k + 1$

In this algorithm schema, the mechanism consists in evaluating the characterization of all sequences of categories. The specificity of selection constraints is used as a control device: when a selection constraint is satisfied, the described category $XP$ is instantiated and the related set of constraints $\Sigma$ is activated. It is interesting to notice that a syntactic category can be projected by any selection constraint, independently from any constituency information (especially the head). Each new assignment, built by adding new juxtaposed categories to the initial set, is then evaluated. Such a completion of the initial assignment is possible when the satisfiability of $\Sigma$ for this new assignment is *acceptable* (cf. line 7). This notion implements the parser flexibility. When we need to build only grammatical structures, *acceptability* is reduced to satisfiability. But for more flexible parsing needs (e.g. spoken language), constraints can be relaxed. The set of constraints to be relaxed, their number, etc. is indicated at this point. Finally, the general process is repeated until no new category can be added.

This parsing schema proposes a general framework in which constraints can be integrated. Each property is implemented by a constraint solver. The mechanism consists in building a characterization for each possible assignment (i.e. any subset of categories). The particularity of selection constraints plays an important role in this schema. In a classical bottom-up technique, the mechanism consists in finding a *handle* which links a set of categories with a non-terminal.

Such a relation in our approach is established between a set of properties and a category. In contrast with phrase-structure techniques, the notion of constituency does not play any particular role. As soon as a selection constraint is evaluated, the corresponding syntactic category is added to the set of categories and all the constraints participating in its description are activated. Concretely, all the selection constraints can be evaluated with no need to know the upper-level category, in contrast to filtering constraints which have to be activated.

Different strategies can be applied according to the needs of the parse. A restricted application stipulates that all constraints have to be satisfied. In this case, only grammatical characterization are built, all ill-formed structures are ruled out. For more flexible applications, typically in the case of parsing spoken language material, constraints have to be relaxed. In this case, characterizations can contain violated constraints. We next describe how CFGs are used in our approach to achieve direct interpretation in these kinds of flexible applications.

## 3 Direct interpretation in CHR

Our methodology for Property Grammar parsing has been designed to provide direct interpretation of Property Grammar rules. This is an interesting contribution with respect to Property Grammars themselves, but also a novel and important proof-of-concept that can lead the way for any constraint-based parsing formalism which relates categories contextually through their properties. To the best of our knowledge, our methodology is the first one that permits an expression of such parsing constraints which is directly and efficiently executable. Thus, we can say that our approach represents for grammars based on contextual properties what DCGs represent for context-free grammars, in the sense that they are as directly executable descriptive formalisms as DCGs[3].

In this section we describe the different components of our methodology : the notion of extended categories, which includes not only traditional information such as category names and features, but also the category's characterization in terms of satisfied and unsatisfied constraints; the modular notation through which a user defines the properties of a given grammar, the single rule through which parsing proceeds, and the analysis of property inheritance which is used in our system.

### 3.1 Extended Categories

Extended categories are of the form: *cat(Name, Features, Graph, Sat, Unsat)* where *Name* is the category name, *Features* a list of features associated with the category (which may be used to check some of the properties between categories), *Graph* is a parse tree which is obtained as a side effect of parsing (which is built

[3] Of course, DCGs do permit context sensitive parsing as well, but the context sensitivity cannot be directly expressed through symbol contiguity, it has to be indirectly expressed in extra arguments or through other extra devices such as linear implication.

even in those cases of incorrect input), and *Sat* and *Unsat* are respectively, the list of satisfied and unsatisfied properties that the immediate daughters of *Name* inside the *Graph* verify between them. In the case of single word categories, the *Sat* and *Unsat* lists will be empty. These categories are created automatically from user's lexical definitions, which are done in terms of CHRG. For instance, a user's entry:

(1) `[the] ::> cat(det,[singular,masculin]).`

compiles into:

(2) `[the] ::> cat(det,[singular,masculin],det(the),[],[]).`

Because these are CHRG rules (i.e., grammar rules, as opposed to plain CHR rules), word boundaries are carried invisibly. If needed, we can retrieve them in a grammar rule by adding `:(Start,End)` after the category, which will unify `Start` to the starting point of the category, and `End` to its end point, or we can write a plain CHR rule that looks at `cat/5` not as a grammar rule, but as the CHR constraint it compiles into, in which case `Start` and `End` can be retrieved as the two first arguments of the corresponding constraint, `cat/7`.

### 3.2 User defined properties

Our system allows the user to enter the specific linear precedence, dependency, requirement, exclusion, constituency and unicity properties that apply to the grammar being defined, through simple primitive predicates which are respectively `prec/3,dep/3,req/3, exclude/3, cons/2`, and `one/2`. Figure 3.2 exemplifies for a simplified noun phrase.

| *Linear precedence* | *Dependence* | *Constituency* |
|---|---|---|
| `prec(det,n,sn).`<br>`prec(det,sa,sn).`<br>`prec(n,sa,sn).` | `dep(det,n,sn).`<br>`dep(n,sa,sn).` | `cons(sn,[det,adj,sa,n]).`<br>`cons(sa,adj]).` |

| *Unicity* | *Exclusion* | *Requirement* | *Phrases* |
|---|---|---|---|
| `one(det,sn).` | `exclude(sa,sup,sn).` | `req(n,det,sn).` | `xp(sn).`<br>`xp(sa).` |

**Fig. 1.** User defined properties

It is to be noted that while the user's definition of constituency is not rigorously needed (since, as we have seen, when selection properties are verified, the determination of constituency follows as a side effect), having it explicitly defined results in improved efficiency. Likewise, phrase definitions can be inferred from any of the other properties, but defining them explicitly makes the system easier and more readable. The user's definitions of properties will be called from system predicates which verify each of these properties on a given set of categories, as we shall see next.

### 3.3 A single rule for inferring all new categories

Conceptually, little more than a single rule is enough for a string's complete bottom-up parse from contiguous constituents. This rule combines two consecutive categories (one of which is of type `XP` or obligatory) into a third, by testing each of the properties on the pair and creating the new property lists through property inheritance (cf. next section). Its form is described in Fig. 2.

```
cat(X,Y,L,TL,RL,SL,UL), cat(Y,Z,R,TR,RR,SR,UR) ==>
    (k_or_xp(R,XP) -> (Ext=L, TE=TR, RE=RR, Sat0=SR, Unsat0=UR);
    (k_or_xp(L,XP) -> (Ext=R, TE=TL, RE=RL, Sat0=SL, Unsat0=UL); fail) ),
    !, ok_in(XP,Ext),
    sat_properties(L,TL,RL,R,TR,RR,XP,Sat0,Unsat0,T,Sat,Unsat)
    | cat(X,Z,XP,TE,T,Sat,Unsat).
```

**Fig. 2.** New Category Inference

This rule tests that one of the two categories (left or right) is a kernel (a phrase head) and the other one of its extension (i.e., complement or adjunct), and then assigns the corresponding features. It then successively tests each of the PG properties among those categories, incrementally building as it goes along the lists of satisfied and unsatisfied properties. Finally, it infers a new category of type `XP` spanning both these categories, with the finally obtained `Sat` and `Unsat` lists as its characterization. In practice, this rule unfolds into two symmetric parts, to accommodate the situation in which the XP category appears before the category Cat which is to be incorporated into it.

Constituents with discontinuities must also be allowed, for completeness. In this case we consider two categories where the end node of the first does not coincide with the start node of the second. Our current research does not yet include discontinuous constituents, for which further linguistic constraints for avoiding combinatorial explosion need to be incorporated.

The parser here described includes as well a sophisticated analysis of types of properties which helps determine how the properties that have been found to hold (fail) in a given phrase are inherited or not by the new phrase comprising that phrase plus a category being incorporated into it. Knowing whether a given property needs to be simply inherited (in which case it just remains as is in the new list of properties) or affects previously recognized properties, and how, allows us an efficient way to move from one list of properties to the next. This analysis is beyond the scope of the present paper. Complete details can be found in [Dahl04b]. The same work proposes a detailed analysis on how to refine the notion of characterization. For our purposes here, let us just say that the user can declare certain properties as relaxed, which results in sentences containing errors related to those properties to be accepted nevertheless, while indication of their failure is given in the output (in the form of a list of unsatisfied properties).

## 4 Extracting selected phrases as a side effect of parsing

One of the interests of our approach lies in its flexibility. The kind of parsing techniques presented above makes it possible to parse substructures as well as subpart of the input. This functionality is useful to deal with part of the input for which no information can be built. It is for example the case with possibly long lists of juxtaposed *NP*, frequent in spoken languages but for which no specific syntactic relation can be given. But it is also interesting for some applications in which the entire syntactic structure or, in other words, the description of all possible syntactic categories is necessary. This is the case for question-answering, information extraction, or terminological applications based on *NP* recognition. In these systems, the knowledge of the argument structure, or the precise description of the relations between the different categories is not necessary. We basically need to know what are the *NP*s of the input and, if possible, their contents. Several techniques can be applied for this (see [Osborne99], [Tjong00]) that usually rely on the specification of patterns defining base *NP*s. In such approaches, it is difficult or even impossible to identify the kind of relations that links the different constituents of the NP. In other words, it is not possible to choose the granularity level. In *PG*, this possibility exists, simply by stipulating the constraints that have to be satisfied among the entire constraint system that forms the grammar. This means that any category can be extracted. Moreover, the description granularity of the extracted category can also be chosen, according to the needs.

### 4.1 Implementation details

Concretely, all irrelevant information is simply not taken into account. For example, the categories that do not participate to a *NP* are ignored. The only special mechanism needed to skip them is a filtering set of predicates which ensures that every category different from *NP* is deleted from the constraint store upon the parse having finished, as first proposed in [Christiansen05a]. For the case of noun phrases, we need to define a constraint we will call cleanup, which we will call after the analysis of any phrase, and, for every category $X$ different from an *NP*, a simpagation rule of the form:

(3) `cat(X, _, _, _, _), !cleanup <:> true.`

The exclamation mark combined with the rewrite symbol shown indicate sympagation in CHRG: once the parse is completed and cleanup is put in the store, the above rule removes from the store the matching category, and the constraint cleanup remains for further use, in order to remove all categories different from np.

The generation of such rules can be automated from a user's command to declare what phrases to focus on, e.g.:

(4) `:-focus(np).`

Similarly, we can include a mechanism for only leaving the outermost *NP*s in the case of embedded ones. This option can be specified by the user through the command

(5) `:−outermost(np).`

It is also possible to imagine an easy reuse from a language to another, simply by adapting the properties. In the case of basic *NP*s, very close properties are used for French and English.

### 4.2 Experiment

We did some experiments in applying this technique to a French medical corpus. The following example illustrates the identification of some noun phrases, extracted together with their syntactic characterizations. In these examples, the output contains the type of the phrase (in this experiment, the noun phrase), its morphosyntactic properties, its constituents and its characterization. This last part of the output, as explained in the second section, is formed by the set of satisfied and violated properties.

The example (6) illustrates the case of a simple *NP*, formed with a determiner and a noun. Its characterization shows for example the satisfaction of a precedence constraint between the determiner and the noun, some uniqueness relations for these constituents as well as a mandatory cooccurrency between the determiner and the noun.

(6)

| | |
|---|---|
| Input | les cellules (*the cells*) |
| Output | `cat(np,[plu,masc], sn(det(les),n(cellules)),`<br>`[prec(det,n), dep(det,n), unicity(det), unicity(n),`<br>`exige(n,det), exclude(name,det), exclude(name,n)],[])` |

The second example, presented in (7), shows the integration of an embedded *AP*. In this example, the *AP* (composed with a single adjective) has been identified separately. The characterization of the *NP* works exactly in the same way as before (as explained in the presentation of the algorithm). In this example too, all constraints belonging to the characterization are satisfied: precedence, exclusion, unicity, etc. One can also remark the stipulation of a dependency relation between the AP and the noun, that illustrates the capacity of the technique to identify syntactico-semantic relations.

(7)

| | |
|---|---|
| Input | les meilleures stratégies (*the best strategies*) |
| Output | `cat(np,[plu,masc], np(det(les), ap(adj(meilleures)),`<br>`n(stratégies)), [dep(ap,n), unicity(n),`<br>`exclude(name,n), exclude(name,ap), exclude(ap,sup),`<br>`prec(det,n), dep(det,n), unicity(det), exige(n,det),`<br>`exclude(name,det)],[])` |

The example (8) presents the case of a complex *NP*. More precisely, this is the case of an ill-formed input, due to a wrong POS-tagging: in this case, the last

adjectives of the list have been tagged as noun instead of adjectives. However, the *NP* has been recognized, due to the possibility of relaxing constraints. In this case, the uniqueness constraint applied to the noun has been violated. This constraint belongs to the second part of the characterization in a separate list of violated properties.

(8)

| | |
|---|---|
| Input | les cellules endothéliales immunotoxines peptides proapoptotiques (*the endothelials ... cells*) |
| Output | `cat(np,[sing,masc],sn(det(les), n(cellules), ap(adj(endothéliales)), n(immunotoxines), n(peptides), n(proapoptotiques)), [prec(det,n), dep(det,n), exige(n,det), exclude(name,det), exclude(name,n), dep(sa,n), exclude(name,sa), exclude(sa,sup)], [unicity(n)])` |

### 4.3 Towards semantics

Using such a symbolic robust approach to *XP* extraction makes it possible to consider integrating semantic information. This is a great advantage in comparison to other methods. Some dependencies can be included in the description so that the extraction can come with different kinds of information, for example concerning the argument structure or the roles of the possible arguments.

We develop in the following the example of *NP* extraction, illustrating how such task is implemented in order to identify *NP*s together with their main modifiers. Basically, the kind of information that is needed in applications mentioned above concerns the noun itself plus its different modifications. This means its quantification (if any) and its classical direct modifiers. In the system described here, the semantic structure of a *NP* is described by its head (the noun itself), its specifier (the determiner) and two possible modifiers (*AP* and *PP*).

(9)
$$\text{SEM}\begin{bmatrix}\text{HEAD } N\\ \text{SPEC } Det\\ \text{MOD } AP\\ \text{COMP } PP\end{bmatrix}$$

Building such structure from a *PG* parser is direct. A minimal set of properties has to be checked in order to identify the *NP* and build the structure. In order to simplify the description and improve the efficiency of the system, we do not take into account relative clauses here. Moreover, also in a simplification perspective, we do not build embedded phrases. The only authorized one is the NP itself. In the end, we obtain a description of the NP which is more complete than the classical definition of a "base NP" (see [Osborne99]) and moreover makes it possible to directly identify the relations inside the *NP*. The properties are then the following:

| | |
|---|---|
| *Constituency* | Const = {N, Det, Adv, Adj, Prep, NP} |
| *Linearity* | Det $\prec$ Adv; Det $\prec$ N; Adv $\prec$ Adj; Adj $\prec$ N; N $\prec$ Prep; Prep $\prec$ NP |
| *Requirement* | Adv $\Rightarrow$ Adj; NP $\Rightarrow$ Prep |

In *PG*, the entire representation of an objects (see [Blache05]) contains on the one hand its properties, and on the other hand, its local information represented in terms of features. The final description of the *NP* we use in the perspective of the *NP* extraction system is then:

(10)

| *NP* | |
|---|---|
| FORM | HEAD *N*<br>SPEC *Det*<br>MOD *Adj*<br>COMP *Prep* |
| PROPERTIES | Const = {Det,N,Adj,Prep, Adv, NP}<br>Det ≺ Adv; Det ≺ N; Adv ≺ Adj; etc.<br>Adv ⇒ Adj; NP ⇒ Prep |

The extraction mechanism consists then in satisfying the set of constraints defined in the property part of the NP object. When, a sequence of words from the input satisfies this set of constraints, the corresponding structure (called FORM) is built. By means of unification, the respective arguments of the structure will be instantiated with the desired values.

## 5 Discussion, Conclusion

In our approach, as we have seen, syntactic categories are inferred from the evaluation of properties, without any need of constituency information.

This aspect has important consequences on the role of constraints in the parsing process. One of the problems with constraint-based approaches is that constraints are usually expressed over high-level objects or structures. This is the case for example in HPSG, in which complex feature-structures must first be built before constraints can be evaluated. Similarly, Optimality Theory also generates a set of structures (or candidate structures) and then uses constraints to filter this set. In our approach, any constraint can be evaluated at any time for any set of categories. Such evaluation, as explained above, dynamically adds new information: the satisfaction of a *selection* constraint instantiates the syntactic category it describes. But this instantiation is conceived almost as a side effect of evaluation: satisfying constraints does not rely on the knowledge of the upper-level category. In other words, the hierarchical information is no longer preponderant in the parsing process. This means that one can evaluate subsets of constraints, for example in the case of applications that only need NP recognition. In this approach, the conception of the relationship between grammar and language becomes very different from that of the generative paradigm. In the latter, a language is conceived as being generated by a grammar. In Property Grammars, a grammar is only used as a characterization device of the language properties.

As a consequence, instead of restricting the role of parsing to the evaluation of the input's grammaticality, we can propose a more flexible vision, in which a

parser's output is the description of all the properties of the input. Concretely, such a description consists in the state of the constraint system after evaluation– in other words, the set of satisfied and violated constraints. We call such state a *characterization* of the input. In some cases, a characterization only contains satisfied constraints, but it can also be the case that some constraints can be violated, especially when parsing real life corpora. In most cases, such violations do not have consequences on the acceptability of the input.

One other formalism that shares the aims and some of the features of Property Grammars are Dependency Grammars (cf. [Tesnière59] and on this point [Mel'čuk88]), a purely equational system in which the notion of generation, or derivation between an abstract structure and a given string, is also absent. However, whereas in Dependency Grammars, as their name indicates, the property of dependence plays a fundamental role, in the framework we are considering it is but one of the many properties contributing to a category's characterization. Perhaps the work that most relates to ours is Morawietz's [Morawietz00], which implements deductive parsing [Shieber95] in CHR, and proposes different types of parsing strategies (including one for Property Grammars) as specializations of a general bottom-up parser. Efficiency however is not addressed beyond a general discussion of possible improvements, so while theoretically interesting, this methodology is in practice unusable due to combinatorial explosion. Moreover, it produces all properties that apply for each pair of categories without keeping track of how these categories are formed in terms of their subcategories, so there is no easy way to make sense of the output in terms of a complete analysis of a given input string.

The idea of throwing away the traditional, hierarchical parsing scheme in favour of a view of parsing which involves properties on categories rather than rewriting schemes first materialized in the 5P formalism (cf. [Bès99a,Bès99b]). Preliminary work re. the advisability of a direct implementation of such an approach had yielded pessimistic results : [Blache95] showed that the mechanism of verification of a constraint system for syntactic analysis could be very expensive, given that the satisfiability of the system had to be verified in each stage. In the present work, however, we have moved beyond that obstacle by our analysis of property inheritance, which removes the need to recalculate all properties at each stage, allowing us to inherit at each stage most of the previous stage's properties, while calculating only the minimally necessary new properties and updating the previous properties along the lines of our property inheritance analysis. Thus, our work has validated the model of property-centered parsing with respect to efficiency, while preserving the level of generality of this theory. In addition, a direct interpretation guarantees a better evolution of the initial system: it can better adjust to changes in the theory and to experimental stages.

We hope to have convincingly argued that direct renditions of flexible, constraint based parsing formalisms can be made to run efficiently while preserving a one to one correspondence between the conceptual and the representational levels, including for such non traditional formalisms as Property Grammars, in

which category inference does not depend on hierarchical or even constituency notions.

The representations allowed by our methodology, while extremely close to the computer-independent, conceptual representations of these formalisms, are directly executable, and moreover non-deterministic. This is satisfying with respect to logic programming's original aims of declarativeness and higher level expressiveness. Together with the advantages of this approach, we are able to even produce hierarchical depictions of the parse history of any category, including "incorrect" or incomplete ones. This is not important in itself, but is provided as an easy side effect, in the interest of historic comfort : we are all used to thinking in terms of parse trees or graphs, so showing a parse record in graph form may prove convenient to some users.

It is interesting to note that the parsing methodology we describe here has been generalised into a concept formation system which provides a cognitive sciences view of problem solving [Dahl04c].

## Acknowledgements

This research was made possible by V. Dahl's NSERC Discovery and Equipment grants and P. Blache's CNRS-SdI grant. Thanks are due to Baohua Gu for his help with testing and debugging the parser.

## References

[Abdennadher98] Abdennadher S. and Schütz H. (1998) "CHR: A flexible query language", in proceedings of *Int. Conference on Flexible Query Answering Systems*, volume 1495 of LNCS, Springer-Verlag.

[Barranco05] Barranco-Mendoza. A. (2005) *Stochastic and Heuristic Modelling for Analysis of the Growth of Pre-Invasive Lesions and for a Multidisciplinary Approach to Early Cancer Diagnosis*, PhD Dissertation, Simon Fraser University.

[Bès99a] Bès G & P. Blache (1999) "Propriétés et analyse d'un langage", in proceedings of *TALN'99*.

[Bès99b] Bès G., (1999) "La phrase verbale noyau en francais". In Recherches sur le francais parlé, GARS, 15, 273-358.

[Blache95] Blache P. & N. Hathout (1995) "Constraint Logic Programming for Natural Language Processing",in proceedings of *NLULP'95*.

[Blache01] Blache P. & J.-M. Balfourier (2001) "Property Grammars: a Flexible Constraint-Based Approach to Parsing", in proceedings of *IWPT-2001*.

[Blache05] Blache P. (2005), "Property Grammars: A Fully Constraint-Based Theory", in H. Christiansen & al. (eds), *Constraint Solving and NLP*, Lecture Notes in Computer Science, Springer.

[Christiansen01] Christiansen, H. (2001) "CHR as grammar formalism, a first report", Sixth Annual Workshop of the ERCIM Working Group on Constraints.

[Christiansen02a] Christiansen, H. (2002) *CHR Grammar web site*, http://www.ruc.dk/~henning/chrg

[Christiansen02b] Christiansen, H. (2002) "Logical Grammars Based on Constraint Handling Rules", in Proc. *18th International Conference on Logic Programming*, Stuckey, P. (ed.) Lecture Notes in Computer Science, 2401, Springer-Verlang, p. 481 .

[Christiansen02c] Christiansen, H. "Abductive Language Interpretation as Bottom-up Deduction", in Proc. *NLULP 2002, Natural Language Understanding and Logic Programming*, Wintner, S. (ed.), Copenhagen, Denmark, pp. 33–47.

[Christiansen05a] Christiansen, H. (2005) "CHR Grammars" in the *International Journal on Theory and Practice of Logic Programming*, special issue on Constraint Handling Rules, pp. 227-248.

[Christiansen05b] Christiansen, H. and Dahl, V. (2005) "HYPROLOG: a new logic programming language with assumptions and abduction", in Proceedings *ICLP'05 (International Conference on Logic Programming*, Sitges, Spain.

[Dahl97] Dahl, V., Tarau, P. and Li, R. (1997) "Assumption Grammars for Natural Language Processing". in *Lee Naish (ed.) Proc. Fourteenth International Conference on Logic Programming*, MIT Press, 1997.

[Dahl04a] Dahl V. "Treating Long-Distance Dependencies through Constraint Reasoning" (2004) in proceedings of *First International Workshop on Constraint Solving for Language Processing(CSLP'04)*.

[Dahl04b] Dahl, V. and Blache, P. (2004) "Directly Executable Constraint Based Grammars", in Proc. *Journees Francophones de Programmation en Logique avec Contraintes*, Angers, France, 149–166.

[Dahl04c] Dahl V. and Voll K. (2004) "Concept Formation Rules: an executable cognitive model of knowledge construction", in proceedings of *First International Workshop on Natural Language Understanding and Cognitive Sciences*, INSTICC Press.

[Dahl04d] Dahl, V. and Tarau, P. (2004) "Assumptive Logic Programming", in Proc. *ASAI 2004*, Cordoba, Argentina.

[Dalrymple91] Dalrymple, M., Shieber, S., and Pereira, F. (1991) "Ellipsis and Higher-Order Unification", In *Linguistics and Philosophy*, 14(4), 399–452

[Frühwirth98] Frühwirth T. (1998) "Theory and Practice of Constraint Handling Rules", in *Journal of Logic Programming*, 37:1-3.

[Maruyama90] Maruyama H. (1990), "Structural Disambiguation with Constraint Propagation", in proceedings of

[Mel'čuk88] Igor Mel'čuk (1988) "*Dependency Syntax*", SUNY Press.

[Morawietz00] Morawietz F. (2000) "Chart parsing as contraint propagation", in proceedings of *COLING-00*.

[Pollard94] Pollard C. & I. Sag (1994), *Head-driven Phrase Structure Grammars*, CSLI, Chicago University Press.

[Osborne99] M. Osborne (1999) "MDL-based DCG Induction for NP Identification", in proceedings ov *CoNLL-99*

[Prince93] Prince A. & Smolensky P. (1993), *Optimality Theory: Constraint Interaction in Generative Grammars*, Technical Report RUCCS TR-2, Rutgers Center for Cognitive Science.

[Shieber95] Shieber S., Y. Schabes & F. Pereira (1995) "Principles and implementation of deductive parsing", in *Journal of Logic Programming*, 24(1-2):3-36.

[Tesnière59] Tesnière L. (1959) *El/'ements de syntaxe structurale*, Klincksieck.

[Tjong00] Tjong Kim Sang E., W. Daelemans, H. Déjean, R. Koeling, Y. Krymolowski, V. Punyakanok & D. Roth (2000) "Applying System Combination to Base Noun Phrase Identification", in proceedings of *COLING-00*.

# DyALog: a Tabular Logic Programming based environment for NLP

Éric Villemonte de la Clergerie

INRIA - Rocquencourt - B.P. 105
78153 Le Chesnay Cedex, FRANCE
Eric.De_La_Clergerie@inria.fr

**Abstract.** We describe DyALog, an environment aimed to compile and run tabular-based parsers for various unification-based formalisms used for Natural Language Processing. Besides providing the full power of logic, DyALog has been enriched with many features to facilitate grammar development and to improve parsing efficiency and robustness. DyALog has also been designed from scratch to provide a very low level integration of tabulation, leading to an original abstract machine.

## 1 Introduction

Historically, Logic Programming has been first developed to process Natural Language, in particular for Parsing. However, at some point, a divergence has occurred with the development of specific Parsing techniques on one side and the development of a complete programming language (Prolog and variants) on the other side.

From the point of view of the Parsing community, the main drawback of most Prolog-like implementations is their inability to handle large amounts of ambiguities as found in human languages. One can also regret the lack of adequate data structures for designing grammars. In particular, Feature Structures [1], typed or not, have become ubiquitous for most unification-based grammatical formalisms, like LFGs and HPSGs. The last point worth mentioning is the lack of simple mechanisms for tuning parsers, for instance to change the parsing strategies or to increase robustness. However, we have to say that many parsers also miss such flexibility.

Still, despite these limitations, Logic Programming remains an interesting paradigm for parsing because of declarativity, important to design and maintain large grammars, because of unification (for handling underspecification), and because of its power as a programming language.

Originally developed to bring to Logic Programming the notion of computation sharing through the integration at a low level of the tabulation techniques found in chart parsers [2], the DyALog system has then evolved towards the creation of efficient parsers for unification-based grammars. While preserving the power of logic programming, it has been extended both internally and at the syntactic level to facilitate the design of grammars for various formalisms and to facilitate the tuning of the resulting parsers in order to increase their efficiency.

The last ten years have seen the emergence of Prolog systems enriched with tabulation (or tabling or memoizing) functionalities, with for instance XSB [3] and B-Prolog. Still, these systems, in contrast to DyALog, are not specifically oriented towards NLP, do not always implement a full tabular system, and generally do not integrate tabular techniques at a very low level. In particular, one may mention that DyALog has replaced the usual copying mechanism of most Prolog systems by an original structure-sharing mechanism, in order to handle the tabulation of large and deep terms as found, for instance, in HPSG grammars. The indexing mechanism, an essential component for efficiently retrieving tabulated terms, has also been extended beyond the usual scheme based on the predicate name and the nature of the leftmost argument. These choices have led to the development of a virtual machine specific to DyALog.

Section 2 presents some syntactic extensions present in DyALog and useful to design grammars. Several grammatical formalisms covered by DyALog are briefly presented in Section 3. The two following sections (Sections 4 and 5) present several functionalities useful to exploit DyALog-based parsers and to increase their efficiency. Section 6 presents the main lines of the architecture of DyALog. Section 7 provides some elements of information about some programs and parsers developed with DyALog, in particular in terms of performance.

## 2 Easing grammar writing

Computational linguists make heavy use of two important data structures, namely Feature Structures and Finite Sets, that have counterparts in most programming languages but not in Prolog. The syntax of DyALog programs has been extended to cover these structures, backed up by efficient implementations.

### 2.1 Feature Structures

Feature Structures [FS] are records with named fields, called features. They allow a more compact and easy writing of terms because there is no need to mention missing features and because feature ordering is irrelevant. A main distinction may be made between *open* FSs vs *closed* FSs. Any feature may occur in an open FS while a closed FS is associated to a *type* that allows a fixed and finite set of features. DyALog only implements closed Typed Feature Structures [TFS] as illustrated below with, first, a declaration of the features allowed for type "np", and then, a FS built on this type.

```
:-features([np],[gen,num,pers,restr,wh]).
lexicon('Sabine', np{ gen => fem,   num => sing,
                      restr => plushum,   wh => (-)   }).
```

TFSs are implemented as standard terms, with, for each type $\tau$ (seen as a term functor), a fixed rank associated to each feature allowed for $\tau$. In consequence, there is no overhead for using these TFSs, both in terms of space and

time. It may be noted that DyALog provides a notion of *namespace* on functors and that FS types are embedded in their own private namespace, to avoid conflicts with other functors.

Following [1], TFSs may be extended by type inheritance, i.e., types may have subtypes and the unification of two TFSs of type $\tau_1$ and $\tau_2$ may result in a new TFS of type $\tau_3$, the most general subtype of both $\tau_1$ and $\tau_2$. A type hierarchy is used to introduce types and to specify subtype relations. Such a hierarchy also lists the features introduced by each type and, possibly, the most general type allowed for a feature in the context of a given type. Features introduced by a type are inherited by all its subtypes, but a subtype may further instantiate the most general type attached to an inherited feature. Such a type hierarchy, as understood by DyALog, is provided below, with the keyword **sub** (resp. **intro**) used to introduce subtypes (resp. features). The notion of inheritance hierarchy has been slightly extended with the keyword **escape**, used to map a type to some of the builtin types of DyALog, namely `symbol`, `integer`, `char` and `compound`.

```
bot sub [string,list,cat,synsem].
  string escape symbol.
  cat sub [s,np,vp,det,n].
    s sub []. np sub []. vp sub [].
    det sub []. n sub [].
  synsem sub [frase,lexeme]
         intro [cat:cat].
    frase sub [root] intro [args:list].
      root sub [] intro [cat:s].
    lexeme sub [] intro [orth:string].
  list sub [ne_list,e_list].
    ne_list sub [] intro [hd:bot,tl:list].
    e_list sub [].
```

A type hierarchy actually holds a lot of default information, that may be used to write TFSs in even more compact ways. For instance, a most general term may be associated by default to each type by recursively applying the type constraints on its allowed features. For a given feature $f$, there is also a most general type introducing $f$ and therefore a default most general term. The unification of two TFSs has now to find their most general subtype, add the newly introduced features, and perform type instantiations on features re-introduced by subtypes. Given a type hierarchy, a small auxiliary program (written in DyALog and Perl) is used to compile all pertinent information into a C library that may then be dynamically loaded by any DyALog binary. In particular, the C library includes a specialized unification function for each pair of unifiable types. The same holds for the subsumption operation. It also includes the code for building the default maximal terms for each type.

The "compilation" of a type hierarchy greatly reduces the time overhead due to unifications and creations of new TFSs. TFSs built on *maximal* types (i.e., types without subtypes) are represented by standard terms as above described, with no overhead. TFSs built on non-maximal terms are internally represented by *dereferencable terms* [dterms] (see Section 6.6) $\tau(X, v1, \ldots, vN)$ where the extra argument $X$ is used to chain subtype instantiation. One may therefore have to follow a chain of such variables to find the current instantiation of a term, hence inducing some slight overhead, both in time and space.

However, in most systems implementing TFSs, a main cause of inefficiency comes from the fact that new terms have to be built during unification, in particular for newly introduced features, and that most of these costly creations become useless because of the high rate of unification failures. DyALog avoids this problem by being based on structure sharing (see Section 6.4) instead of structure copying as most systems.

### 2.2 Finite Sets

Finite Sets [FSets] terms are finite collections of atomic values and are generally used to encode the many collections of such values found in Computational Linguistics, for instance the mood values for verbs. In DyALog, the list of atomic values associated with a functor must first be declared, as shown below.

```
:-finite_set(mode, [ind,subj,inf]).
tag_lexicon(aime, '*AIMER*', v, v{ mode => mode[ind,subj], mum => sing }).
```

FSet terms are implemented as dterms of the form $f(X, BV_1, \ldots, BV_n)$ where $BV_i$ denotes a bit vector. Unification is very efficiently performed by applying a logical AND instruction between corresponding bit vectors. A FSet term may be maximally instantiated to a single constant.

FSet terms are very useful to handle, in an efficient way, ambiguities on atomic values, but also, like closed Feature Terms, to avoid typos when developing a large grammar. Indeed, the use of a non-allowed value in a given set (like the use of a non-allowed feature) raises an error at compilation time.

### 2.3 Miscellaneous extensions

The syntax of DyALog also provides some other minor extensions. The first one is the notion of *immediate unification* `::` which (while slightly evil) is very useful to handle the duplication of complex terms, especially dterms, as found, for instance, in HPSG-like grammars. A term p(X**::**f(Y),X) is actually equivalent to either p(f(Y),f(Y)) or more precisely to p(X,X),X=f(Y), except that the binding of $X$ to $f(Y)$ is performed when reading the terms.

Inspired by HPSG, the following example combines the use of immediate unification with the use of default notations for TFSs made possible by type hierarchies. In this example, the variable `NP` is immediately bound to the most general term $t$ that may introduce a feature `cat` and $t$ will also be propagated to the first occurrence of `NP` in the clause head.

```
hpsg(root{} :: args => ne_list{ hd => NP,
                                tl => (hd => VP :: tl => e_list{}) })
    --> hpsg( NP :: cat => np{} ), hpsg( VP :: cat => vp{} ).
% next is compact form for lemma(lexeme{orth=>le,cat=>det{}}).
lemma(orth => le :: cat => det{}).
```

The second extension is mostly used to get a (pseudo) flavor of higher order terms, with *Hilog* terms like p(a,b)(1,2), internally represented by apply(p(a,b),1,2). The Hilog notation is very convenient when one has to handle several sets of arguments (see Section 3.4) or to express semantic formula, such as P^X^Y^P(X,Y).

## 3 Grammatical Formalisms

DyALog covers most standard functionalities of Prolog systems[1], including usual builtins and the ability to bind 'C' functions. However, the main richness of DyALog comes from its coverage of several grammatical formalisms, besides the commonly available Definite Clauses Grammars [DCGs].

### 3.1 DCGs

The implementation of DCGs [4] is standard and provides most of the expected functionalities, including the possibility, shared by all other formalisms, to escape to logical predicates, through the use of the curly operator {Goal}. Still, DCGs have been slightly extended with the possibility to use the operators +> and <+ for guiding the direction of parsing. For instance, "a −−> b <+ **c** +> d +>e." will start recognizing $c$, then recognize $d, e$ on the right of $c$ and then $b$ on the left of $c$. DCGs have also been extended by the various generic functionalities listed in Sections 4 and 5.

### 3.2 BMGs

Bound Movement Grammars [BMG] are a variant of extraposition grammars and are implemented as an extension of DCGs. The basic idea of BMGs is that constituents may be temporarily pushed on some stack to be discharged in some other place to fill a trace. Island constraints may be used to block the displacement of constituents inside some non-terminals. As illustrated by the following example, the directives "bmg_stack", "bmg_pushable", and "bmg_island" are respectively used to define stacks (here we have 3 stacks for handling topicalization, relatives, and interrogatives), non-terminals allowed on stacks, and island operators. The clause on line 8 may be used to push a topicalized constituent pp on the **slash** stack to discharge it in a verbal group (line 9) but not in a subject nominal group because of the island barrier **isl_slash** (line 6).

```
:−bmg_stacks([slash,rel,wh]).
2 :−bmg_pushable(np,[wh,rel]).
:−bmg_pushable([v,pp],[slash]).
4 :−bmg_island(isl_relwh,[rel,wh]).
6 s −−> isl_slash np, vp.
  s −−> comp, s.
8 comp slash pp −−> isl pp.
  vp −−>v,np,pp.
```

### 3.3 Feature TAGs and TIGs

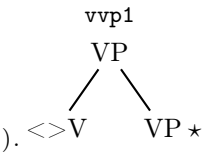
Tree Adjoining Grammars [TAG] [5] are actually the main grammatical formalism explored with DyALog. A TAG is formed by a set of elementary initial and auxiliary parse trees that may be combined to form complete derived parse

[1] excluding a full implementation of the cut operator, replaced by a more restricted and local notion of if-then-else construction.

trees using *substitutions* and *adjunctions*. A substitution replaces a leaf node labeled by a non-terminal $N$ by an *initial* elementary tree $\alpha$ whose root label is $N$. An adjunction inserts an *auxiliary* elementary tree $\beta$ with root label $N$ on some internal node $\nu$ labeled by $N$, attaching the subtree rooted below $\nu$ on the distinguished *foot* node of $\beta$. Feature TAGs extend TAGs by decorating nodes by pairs of `top` and `bottom` arguments, usually expressed by Feature Structures.

Partly for efficiency reasons, and partly for linguistic reasons, TAGs are usually *lexicalized*, i.e., with at least one lexical leaf in each tree. However DyALog can handle both lexicalized and non-lexicalized trees, nevertheless exploiting lexicalization when possible (see Section 5). In a non-mandatory way, grammars can be organized following the XTAG model [6], where (a) each tree has an *anchor* node, (b) trees are grouped into families sharing common linguistic properties (such as a sub-categorization frame), (c) word-forms refers to lemmas, and (d) lemmas indicate which families they may anchor. A lemma may add additional constraints on the lexical values or on the arguments of the nodes of the anchored trees. For instance, the following example shows how the auxiliary tree `vvp1`, belonging to the family `vvp`, which is anchored by modal verbs such as "POUVOIR" (*to be able*). This lemma adds the additional constraints that the node `VP` should be an infinitive.

```
tag_tree{ name => vvp1, family => vvp,
          tree=> auxtree bot=VP::vp{}
                 at vp( <> v, id=vp_ and bot=VP at *vp) }.
tag_lemma('POUVOIR',v,
    tag_anchor{ name=>vvp,
                equations=>[bot = vp{ mode=>inf } at vp_]}).
tag_lexicon(peut, 'POUVOIR', v, v{ mode => ind, num => sing }).
```




Anchoring by family represented by atomic values is too rigid in practice, for at least two reasons. First, in the context of large coverage grammars with several thousand trees, several hundred families, and several tens of thousands of lemmas, one has to be sure that the same name is used in a correct way in both the grammar and the lexicon. A modification of the grammar (addition, modification or deletion of families) has to be reflected by many changes in the lexicon. Secondly, a tree may be common to several families, hence potentially leading to the duplication of trees.

To overcome this rigidity, DyALog allows the use of feature structures called *hypertags* in the place of atomic family names. The features of an hypertag correspond to a small set of linguistic properties that are easily identified and stabilized. The hypertag $\mathcal{H}_\tau$ attached to a tree $\tau$ specifies which syntactic properties are handled by $\tau$ while the hypertag $\mathcal{H}_l$ attached to a lemma $l$ specifies the syntactic properties of $l$. The anchoring of $\tau$ by $l$ is only possible if the hypertags $\mathcal{H}_\tau$ and $\mathcal{H}_l$ do unify.

Tree Insertion Grammars [TIGs] [7] are a restriction of TAGs where the auxiliary trees only insert material on the left or on the right sides of adjunction nodes, but not simultaneously on both sides. This property implies that the foot nodes are either the leftmost or the rightmost leaves of auxiliary trees. The interest of TIGs (without Features) is that they are actually strongly equivalent

to CFGs and parsable in $O(n^3)$ instead of $O(n^6)$ for TAGs. Furthermore, TAGs for human languages (at least for French and close languages) are essentially TIGs. Actually, DyALog can handle hybrid Feature TAG/TIG grammars and analyze a TAG grammar to identify its TIG left and right auxiliary trees.

### 3.4 Feature RCGs

Range Concatenation Grammars [8] form a very wide and powerful class of grammars that may however still be parsed in polynomial time. In particular, Mildly Context-Sensitive Formalisms, including TAGs, can be encoded through RCGs. RCGs clauses are similar to DCGs clauses with the main difference that the arguments of non-terminals actually specify ranges over the input string. These arguments are either terminal values or variables, possibly satisfying concatenation constraints, expressed with the operator @. DyALog extends RCGs by allowing a second set of logical arguments, besides the first set of range arguments, relying on Hilog terms. For instance, the following RCG recognizes the language $a^n b^n c^n$, using an extra logical argument for counting and returning $n$.

```
s(N) (X @ Y @ Z) --> a(N) (X,Y,Z).                    a(0) ("","","") --> true.
a(M) ("a" @ X,"b" @ Y,"c" @ Z) --> a(N) (X,Y,Z), {M is N+1}.    axiom(s(N)).
```

## 4 Writing factorized grammars

Large coverage grammars, and particularly lexicalized (or lexicalizable) ones tend to be very large, with for instance several thousand trees for wide coverage TAGs, raising maintenance problems and parsing efficiency issues. Actually, the size of such large grammars mostly arises from redundancies. For instance, in a TAG, most verb-anchored trees provide a subject node decorated by agreement conditions with the verb, with various realizations and positions.

DyALog implements several factorization operators, generic in the sense that they can be used for most grammatical formalisms. For DCGs and BMGs, they apply on literals or sequences of literals. For TAGs and TIGs, they apply on nodes or, more generally, on sequences of sibling nodes.

The first operators are the well-known *disjunction*, *optionality* (with $x? \equiv x;\epsilon$), and *Kleene star* operators. The Kleene star operator may be completed by explicit minimal and maximal bounds, and by aggregation predicates (for instance, to build a list of arguments through the loop).

As syntactic sugar for optionality combined to Prolog escaping, and for a finer control of optionality, the notion of *guards* (for TAGs) may be used to state conditions on the presence or absence of a node (or of a node sequence). An expression $(G_+, x; G_-)$ means that the guard $G_+$ (resp. $G_-$) should be satisfied for $x$ to be present (resp. absent). A guard $G$ is a Boolean expression on equations between FS paths and is equivalent to a finite set of substitutions $\Sigma_G$.

Less known but well motivated in [9] to handle local free word ordering, the *interleaving* (or shuffling) of two sequences $(a_i)_{i=1\cdots n}\#\#(b_j)_{j=1\cdots m}$ returns

all sequences containing all $a_i$ and $b_j$ in any order that preserves the original orderings (i.e., $a_i < a_{i+1}$ and $b_j < b_{j+1}$).

All these operators do not increase the expressive power or the worst-case complexity of the underlying formalism. Indeed, they can obviously be expanded and removed by adding new non-terminals and new trees to the grammar. For instance, a tree $\tau[t_1; t_2]$ with some occurrence of a disjunction $(t_1; t_2)$ may be replaced by the trees $\tau[t_1]$ and $\tau[t_2]$. A tree $\tau[(G_+, t; G_-)]$ may be replaced by the set of all trees $\tau[t_1]\sigma_+$ and $\tau[\epsilon]\sigma_-$ with $\sigma_+ \in \Sigma_{G_+}$ and $\sigma_- \in \Sigma_{G_-}$. However, the number of added productions may be exponential in the number of operators in a given production.

These operators are implemented without expansion, ensuring good performances and more natural parsing outputs (with no added non-terminals in the derivation forests, Section 5.1). Care has been taken of several issues related to logical variables (for instance to have shared or fresh variables between each iteration of the Kleene star operator) and interactions between operators (in particular between the interleaving and Kleene star operators).

DyALog also provides an intersection operator "&" which does change the expressive power of the underlying formalism but does not increase its worst-case complexity. For instance, the non-CFG language $a^n b^n c^n$ is recognized with the production "s −−> 'AnBnC∗' & 'A∗BnCn'". This intersection operator looks promising to handle some non-linear phenomena that can be expressed by the conjunction of several properties on a same segment of the input string.

## 5 Tuning Parsers

As input, DyALog parsers take either lists, as usual for Prolog systems, or, more efficiently, ambiguous word lattice (or DAGs) coded by 'C'/3 facts, as shown below for the sentence "*[unknown word] watches [unknown words] with a telescope*".

```
'C'(0,_,1). 'C'(1,watches,2). 'C'(2,_,2). 'C'(2,with,3). 'C'(3,a,4). 'C'(4,telescope,5).
```

The reading of terminals (in TAGs and DCGs) and the anchoring of trees may be customized, to be able, for instance, to make a call to a lexicon or check hypertags. Terminals and anchors may also be used for grammar filtering: the grammar is compiled offline but only part of the compiled grammar is activated depending on the words of the input lattice. It should be noted that non-lexicalized productions are allowed and are systematically activated.

Default loading conditions are associated to each kind of productions, but it is also possible to specify its own, as shown below:

```
'$loader'(phrase([qui],_,_), (np --> np, [qui], srel)). % '$loader'(Cond,Clause).
```

A common complaint against deep parsers is their lack of robustness and their inability to return information when failing a parse. In a way, this is partly a wrong issue concerning DyALog parsers, because it is immediate to switch from full to partial parsing, by generalizing the topmost query over the position variables $L$ and $R$.

```
?–recorded('N'(R)), L=0, tag_phrase(s,L,R). % Full parsing
?– tag_phrase(s,L,R) ; tag_phrase(np,L,R). % Partial Parsing
```

Because the computations are tabulated (see Section 5.3), partial parsing may be completed by a post parsing call to a merging and filtering predicate over partial parses to retrieve the best sets of partial parses covering the input.

Finally, it is worth noting that DyALog provides various levels of tabulation for predicates and, in a lesser measure, for non-terminals. By default, literals are (strongly) tabulated but the following specifications can also be used, and may be much more efficient in some occasions.[2] predicate or non-terminal declared as **std_prolog** or **rec_prolog** are not tabulated, on the condition that all its descendants are not or only weakly tabulated.[3] A **light_tabular** predicate is weakly tabulated, in the sense that it can be computed without entering a loop, in itself or one of its descendants[4]. A **prolog** predicate is not tabulated, except indirectly through the possible tabulation of one of its descendants.

### 5.1 Shared Derivation Forests

All DyALog parsers can return, using the option `-forest`, the ambiguous set of derivation trees under the form of a compact *shared derivation forest* [10]. As illustrated below for a TAG derivation of sentence "Yves loves Sabine", a shared forest is equivalent to a CFG, where the "non terminals" correspond to grammatical operations (for instance, substitutions, adjunctions and anchoring for TAGs) completed by their span and by feature instantiation, while the "productions" indicate the possible derivations for the elements intervening in an operation.

```
s{inv=> −, mode=> ind}(0,3)          1 <— [subject]2 [<>]3 [object]4
np{gen=> masc, num=> sing }(0,1)     2 <— [<>]5
tag_anchor(loves,1,2,tn1)            3 <—
np{gen=> fem, num=> sing }(2,3)      4 <— [<>]6
tag_anchor(Yves,0,1,np)              5 <—
tag_anchor(Sabine,2,3,np)            6 <—
```

Ambiguity is represented by disjunctions inside the productions of the forest and sharing by the reuse of "non terminals", as shown below (where a prepositional attachment 5 can be either performed on the verb or on the object):

```
s{}(0,7)  1 <— ( [subject]2 [<>]3 [object]4 [vp]5
               | [subject]2 [<>]3 [object]6 )
```

The (non mandatory) labels such as `subject` and `object` come from labels attached to the non-terminals occurring in the grammar productions. Originally introduced for TAGs, the idea has been extended for all formalisms, using "tagop" to define its own labeling operator, as shown below for DCGs.

[2] Unfortunately, the DyALog compiler does not (yet) perform an automatic analysis to select the best choice.

[3] **std_prolog** predicates must be defined by a single clause and can be internally compiled by a 'C'-like function.

[4] The consequence is that there is no need to setup a costly mechanism to propagate answers trough loop points until reaching a fix-point.

:−tagop(':'). s —> subject:np, verb:v, object:np.

Labels are very useful to decode forests by decorating them, for instance, with function labels. They are also helpful to setup post-parsing forest traversals to disambiguate them or to extract information (see Section 5.3).

Forest are extracted by following typed back-pointers attached to tabulated objects. The derivation of some objects may even disappear or be inlined in the derivation of an other object (for instance, for the derivations of the pseudo non-terminal introduced for Kleene stars). Labels are stored inside the back-pointers.

For readability, transformations may be applied to the grammar representation of shared derivation forest to get more graphical representations, with, in particular, a dependency based one for TAGs.[5]

### 5.2 Parsing strategies

Modulation [11] on non-terminals and their arguments may be used to specify which information should be exploited during the top-down prediction phase, the rest being checked during the bottom-up answer propagation phase. For instance, the declaration dcg_mode(np/2,+(−,−),+,−) states (with the + marks) that only the terminal name `np` and its left position (in the input string) should be used for prediction, the right position and its two arguments being checked on the propagated answers. Using modulation, it is possible to get various customized parsing strategies ranging from a pure top-down one (full prediction) to a pure bottom-up one (no prediction).

The bidirectional parsing strategy mentioned for DCGs has been used as a base, through grammar rewriting, for head-driven parsing [12]. Left-Corner based parsing strategies are also possible for DCGs, taking into account the modulation information for computing the left-corner relation. LC parsing strategies were originally proposed for CFGs, and while interesting, their transposition to DCGs shows that some efforts have still to be done to increase their efficiency.[6].

We are also investigating the use of *guiding techniques* to speed up parsing. The basic idea is to compute some over-generating approximation of the original grammar, parse the input string with it, and use the information of the resulting shared forest to guide the predictions of the original grammar. For instance, we have tried approximating Feature TAG grammars by TIGs without Features. One could also approximate DCGs or TAGs by simpler CFGs, or even regular approximations.

### 5.3 Advanced Tabular-based parsing

DyALog inherits from its logic programming lineage the ability to easily design parser interpreters with just a few tens of clauses. For instance, such parser

[5] They can be tried on line at `http://atoll.inria.fr/parserdemo`.

[6] Essentially, the efficiency of LC strategies for CFGs relies on the use of very efficient look-up tables for the LC relation. For indexing reasons, it is difficult to get the same efficiency for logical terms.

interpreters were originally developed for TAGs and RCGs before moving to a tighter and more efficient integration in the DyALog compiler. The main advantage of DyALog parser interpreters, compared to similar interpreters on top of other Prolog systems, comes from their inheritance of the tabular properties of DyALog. Their main disadvantage, compared to a compiler integration, is the difficulty to extract a shared derivation forest that is not polluted by the predicates of the interpreter.

Tabulation is generally used to achieve computation sharing but it is actually interesting to take benefit of it to go beyond traditional parsing techniques. In particular, DyALog includes predicates to look up for a tabulated object (recorded(O)), to wait for an answer to be tabulated ('$answers'(G) where $G$ can even be a variable) [7] , or to follow back-pointers. That means that it is possible to design dynamic parsing strategies by examining tabulated objects and tabulated derivations (using back-pointers). For instance, we are exploring the handling of coordinations, by waiting for derivations reaching a coordination word $w$ like 'and', and duplicating (with ellipses) these derivation on the right of $w$. Parsing corrections based on tabulated objects can also be performed. Another experiment that is under way is to build semantic forms, just after parsing, by retrieving derivations and transforming them with DyALog predicates.

## 6 Architecture

### 6.1 Tabular-based model

DyALog is based on a tabular model, illustrated on Figure 1, with a table of objects and an agenda. Until the agenda is empty, an object is selected, following a fair policy, and combined with tabulated objects to build new objects. Unless subsumed by already tabulated objects, these new objects are tabulated and scheduled in the agenda.

### 6.2 Compiling

Grammars are compiled with a bootstrapped compiler. The compilation is based on theoretical results about automata, in particular Logical Push-Down Automata (for DCGs) and 2-Stack Automata (for TAGs) [13, 14]. Basically, the transitions of an automaton encode the steps of a parsing strategy for a given grammar. Dynamic Programming [DP] interpretations specify how the automata derivations can be broken into elementary pieces that may be tabulated and combined together to retrieve all derivations. Exploiting these DP interpretations, and given a grammar, the compiler builds the various transitions during a first phase and then, in a second phase, identifies the skeletons of the objects that

[7] If considering the *ask/tell* paradigm of Concurrent Constraint Solving, this meta-predicate '$answers'/1 is actually very similar to an *ask* request to the set of answers, seen as a constraint store. Tabulation may then be seen as "*telling*" answers. In practice, it is indeed possible to use a concurrent programming style with DyALog.

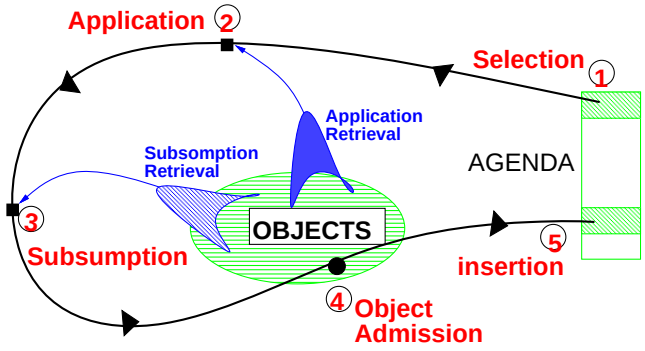



**Fig. 1.** DyALog evaluation loop

may be tabulated and prepares the code allowing the combination of these objects. The third phase emits the instructions for a mini-assembler, specific to DyALog, which is then expanded to a true assembler.

### 6.3 Abstract Machine

Besides the above-mentioned table and agenda, the abstract machine for DyALog includes a heap for allocating terms and objects, and 3 stacks. The Control Stack is used to keep trace of alternatives and continuations and the Trail Stack to keep trace of various information to be undone on backtracking (including variable bindings). The third Layer Stack is used for structure sharing.

### 6.4 Structure Sharing

Almost all WAM-based systems use structure copying to handle the renaming of clauses before using them. While interesting for a statically known restricted set of clauses, copying become less interesting over an arbitrary number of potentially deep tabulated objects. Instead, DyALog exploits a structure-sharing mechanism based on *layers* [15] where renaming is handled with a very low cost. As illustrated on Figure 2, each tabulated object includes an ordered sequence of layers, each layer holding a set of bindings represented by *Virtual Copy Arrays* [VCAs] for fast accesses. A binding $X/t_\delta$ on layer $k$ means that $X$ on layer $k$ is actually bound to term $t_{k+\delta}$. When an object is activated, its layers are pushed on the Layer Stack, this simple operation being sufficient to handle renaming. The temporary bindings created during unification are pushed on the Trail Stack and dereferencing has to consult, in order, both the Trail and Layer stacks. When a new object is tabulated, the pertinent layers and temporary binding are merged, with a minimum of copy, to form a new set of layers.

Each distinct term is uniquely represented in the heap, allowing us to speed up the unification and subsumption operations. For instance, the unification

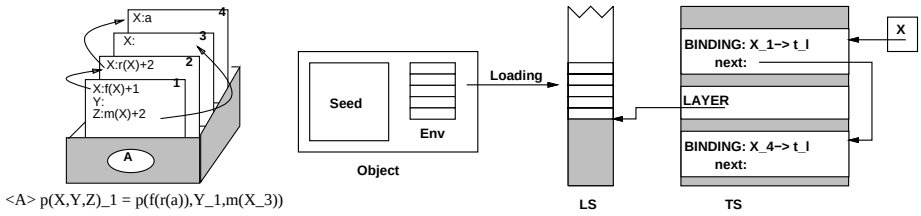



**Fig. 2.** Layer Sharing

of two ground terms, atomic or not, is just a pointer comparison. When two identical terms with variables are unified, the terms are not traversed and the unification is only tried on their variables.

The structure-sharing model used by DyALog, based on the loading and saving of binding environments, is potentially interesting for tabular constraint solving: a saved environment would be a saved constraint store and the saving operation would correspond to the existential closure of a store w.r.t. a set of variables. This closure may however be computationally costly.

### 6.5 Indexing

One of the main critical issues for tabulation-based systems is the cost for retrieving pertinent tabulated objects, that should be low enough to justify tabulation over re-computations. The simple indexing scheme used by most Prolog system is not powerful enough, when the table is very large and hold complex objects. Instead, DyALog indexes the predicate and all its arguments down to some bounded depth, using a trie of hash tables on all these indexing keys [16]. The current implementation is relatively efficient, except in the context of deep terms where the most discriminating arguments are too deep to be indexed. Following current proposals made for HPSG implementations (*quick check filtering*) [17], we plan to complete objects by a small set of indexing keys, identified through the parsing of corpora as being the most discriminative ones (i.e., causing a maximum of unification failures).

### 6.6 Derefencable terms

To handle TFSs that may instantiated to some sub-type, a more generic notion of Dereferencable-Terms (dterms) was introduced. Dterms are specially flagged terms which a special handling of their first argument. If the first argument $t_1$ of a dterm $t = f(t_1, a_1, \ldots, a_n)$ is not a variable, $t_1$ is itself dereferenced and used in place of $t$. An unification involving a dterm $t$ may call a specialized unification procedure depending on the functor of $t$ and this unification may instantiate the first argument of $t$. Non-maximal TFSs and FSet terms are implemented as dterms. DyALog also implements infinite terms as dterms built

on functor $LOOP/3. Dterms look like a promising entry point for a future and more complete handling of typing constraints.

## 7 Experiments

Besides using DyALog to handle itself, we have also used DyALog for small to medium size DCGs and TAGs grammars. DyALog was also used to develop a robust Portuguese parser, based on a first head-driven bi-bidirectional DCGs for non recursive chunks, and completed by a BMG [12].

More recently, DyALog has been used to develop a Meta Grammar Compiler `mgcomp` [18]. The idea of *Meta-Grammar* [MG] [19] is to factorize linguistic information through a multiple inheritance hierarchy of small elementary classes, each of them grouping elementary constraints between nodes (precedence, dominance, equality, ...) and on node decorations. A class may also state that it provides or requires some functionality (for instance, the notion of subject). The first task of of the MG compiler is to cross classes in order to neutralize these required and provided functionalities. Its second task is to check the satisfiability of the constraints accumulated by the neutral classes. Its third task is to use the constraints of the surviving neutral classes to emit minimal TAG trees. The first and second tasks rely on the computation of transition closures for which the weak tabulation mechanism of DyALog proved to be efficient. It may be noted that these 2 tasks correspond to typical constraint solving tasks, namely solving and generating. For handling a larger typology of constraints, it could be interesting to combine or extend DyALog with a true constraint solver.

By combining the possibilities of description factorization provided by MGs and by DyALog, we were able to quickly develop, for French, a highly factorized TAG Grammar of only 133 trees, with only 27 trees anchored by verbs, roughly corresponding to several thousand non-factorized trees. The resulting hybrid TAG/TIG parser was compiled with a left-to-right top-down parsing strategy, modulated not to use the node decorations for predictions.

The parser may be used to try either full or partial parsing, with, in the latter case, the selection of the best covering of the input sentence by partial parses. The coverage rate for full parsing is excellent (over 93%) on the test suites that we have used during the development of the grammar (with 334 sentences for EUROTRA and 1661 for TSNLP), with a lexicon of more than 400000 word forms. The parser has also been tried on a heterogeneous corpus of almost 40000 sentences in the context of a parsing evaluation campaign. We achieved a coverage rate around 42% for full parsing. We are still waiting for the full results of this campaign, but we have computed a preliminary F-measure of 0.71 for recognizing elementary chunks on a reference subset of around 2517 sentences. With a slightly modified and less generating grammar, we very recently achieved a 41% coverage for full parsing on a journalistic corpus of around 330K sentences. Fig. 3 shows the reasonable time distribution we got (on a P4 at 3.2Ghz) w.r.t. the number of edges of the input lattice (with an excellent approximation of 1.17 edges per word) and with a timeout of 50 s.

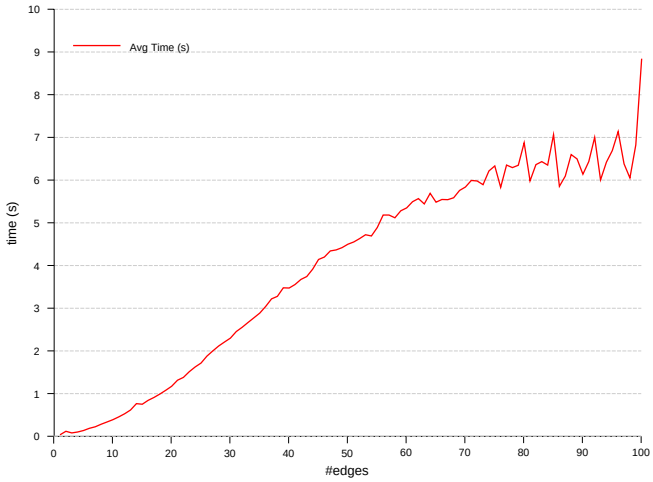



**Fig. 3.** Time distribution over MD 00-09 (timeout=50s)

## 8 Conclusion and evolutions

We see DyALog as a multi-purpose system. It could provide an interesting environment for educational purpose, because it is easy to quickly develop small to medium scale grammars for various formalisms, to test distinct parsing strategies, and to avoid looping and most combinatorial problems thanks to tabulation. DyALog is also interesting for researchers wishing to explore new grammatical formalisms or new parsing strategies, either through the fast development of meta-interpreters or through extensions of the compiler. The concurrent-like programming style provided by DyALog looks like a promising way to handle complex linguistic phenomena, such as coordination and error repairs. Finally, DyALog is now mature enough and powerful enough to handle wide-coverage grammars for practical tasks, particularly when using factorized grammars.

DyALog should be soon extended with Thread Automata [20], for encoding, in an uniform way, parsing strategies for an even wider range of Mildly Context-Sensitive formalisms, including local Multi-Component TAGs. It should also be extended to properly handle probabilities, either as a post-parsing task on the shared forests for disambiguating them, or during parsing to schedule in priority the highest probability computation paths.

DyALog and most of the tools and grammars mentioned in this paper are freely available on `http://atoll.inria.fr` under the "Catalog" entry.

## References

1. Carpenter, B.: The Logic of Typed Feature Structures with Applications to Unification Grammars, Logic Programs and Constraint Resolution. Cambridge University

Press (1992)
2. Earley, S.: An efficient context-free parsing algorithm. In: Communications ACM 13(2). ACM (1970) 94–102
3. Sagona, K., Swift, T., Warren, D.: XSB as an efficient deductive database engine. In: Proc. of SIGMOD'94. (1994)
4. Pereira, F.C.N., Warren, D.H.D.: Definite clause grammars for language analysis - a survey of the formalism and a comparison with augmented transition networks. Artificial Intelligence **13** (1980) 231–278
5. Joshi, A.K.: An introduction to tree adjoining grammars. In Manaster-Ramer, A., ed.: Mathematics of Language. John Benjamins Publishing Co., Amsterdam/Philadelphia (1987) 87–115
6. Doran, C., Egedi, D., Hockey, B.A., Srinivas, B., Zaidel, M.: XTAG system — a wide coverage grammar for English. In: Proc. of the 15th International Conference on Computational Linguistics (COLING'94), Kyoto, Japan (1994) 922–928
7. Schabes, Y., Waters, R.C.: Tree insertion grammar: a cubic-time, parsable formalism that lexicalizes context-free grammar without changing the trees produced. Fuzzy Sets Syst. **76** (1995) 309–317
8. Boullier, P.: Range Concatenation Grammars. In: New Developments in Parsing Technology. H. bunt, j. carroll, and g. satta edn. Volume 23 of Text, Speech and Language Technology. Kluwer Academic Publishers (2004) 269–289
9. Nederhof, M.J., Satta, G., Shieber, S.: Partially ordered multiset context-free grammars and free-word-order parsing. In: In 8th International Workshop on Parsing Technologies (IWPT'03). (2003) 171–182
10. Billot, S., Lang, B.: The structure of shared forests in ambiguous parsing. In: Proc. of the 27 Annual Meeting of the Association for Computational Linguistics. (1989)
11. Barthélemy, F., Villemonte de la Clergerie, É.: Information flow in tabular interpretations for generalized push-down automata. Theoretical Computer Science **199** (1998) 167–198
12. Rocio, V.J., Lopes, G.P., Villemonte de la Clergerie, É.: Tabulation for multi-purpose parsing. Grammars **4** (2001) 41–65
13. Éric Villemonte de la Clergerie, Alonso Pardo, M.: A tabular interpretation of a class of 2-stack automata. In: Proc. of ACL/COLING'98. (1998)
14. Alonso, M.A., Villemonte de la Clergerie, E., Diaz, V.J., Vilares, M. (In:)
15. Villemonte de la Clergerie, É.: Layer sharing : an improved structure–sharing framework. In: Proc. of POPL'93. (1993) 345–356
16. Villemonte de la Clergerie, É.: Automates à Piles et Programmation Dynamique. DyALog : Une application à la programmation en Logique. PhD thesis, Université Paris 7 (1993)
17. Ciortuz, L.: 10. In: On two classes of Feature Paths in Large Scale Unification Grammars. Volume 23 of Text, Speech and Language Technology. Kluwer Academic Publishers (2004) 203–228
18. Thomasset, F., Villemonte de la Clergerie, E.: Comment obtenir plus des méta-grammaires. In: Proceedings of TALN'05, Dourdan, France, ATALA (2005)
19. Candito, M.H.: Organisation modulaire et paramétrable de grammaires électroniques lexicalisées. PhD thesis, Université Paris 7 (1999)
20. Villemonte de la Clergerie, E.: Parsing mildly context-sensitive languages with thread automata. In: Proc. of COLING'02. (2002)

# Lexicalised Configuration Grammars

Robert Grabowski, Marco Kuhlmann, and Mathias Möhl

Programming Systems Lab, Saarland University, Saarbrücken, Germany

**Abstract.** This paper introduces Lexicalised Configuration Grammars (LCGs), a new declarative framework for natural language syntax. LCG is powerful enough to encode a large number of existing grammar formalisms, facilitating their comparison from the perspective of graph configuration. Once a formalism has been encoded as an LCG, the framework offers various means to increase its expressivity in a controlled manner; trading expressive power for computational complexity, this makes it possible to model syntactic phenomena in novel ways. Parsing algorithms for LCGs lend themselves to a combination of chart-based and constraint-based processing techniques, allowing both to bring in their strengths.

## 1 Introduction

Formal accounts of natural language syntax may differ in their understanding of *grammar*. In *generative frameworks*, grammars are systems of *derivation rules*; well-formed expressions correspond to successful derivations in these systems. In *descriptive frameworks*, grammars are complex constraints on syntactic structures; well-formed structures are those that satisfy a grammar. This paper presents Lexicalised Configuration Grammars (LCGs), a new descriptive framework for the syntactic analysis of natural language.

*Structures and constraints* LCG does not replace existing grammar formalisms; it offers a formal landscape into which these formalisms can be embedded to study them and their relations from a different angle: as description languages for syntactic structures. To be expressed as an LCG, a grammar formalism needs to be characterised by two choices: (1) What structures does it describe? and (2) What constraints does it use to describe them? To illustrate this, we will show how context-free grammars (CFGs) fits into LCG.

Following McCawley [1], CFGs can be seen as description languages for ordered, labelled trees (Choice 1). More precisely, let $G = (\Sigma, \Pi, R, \pi)$ be a CFG with $\Sigma$ and $\Pi$ being the alphabets of terminal and non-terminal symbols, respectively, $R$ the set of rules, and $\pi \in \Pi$ the start symbol. A node $u$ *satisfies* $G$ if either (a) $u$ is a leaf node and is labelled with a terminal symbol, or (b) $u$ is an inner node with successors $u_1, \ldots, u_k$ (in that order), $R$ contains a rule $\alpha \to \beta_1 \cdots \beta_k$ (where $\alpha \in \Pi$ and $\beta_i \in \Sigma \cup \Pi$), $u$ is labelled with $\alpha$, and each successor $u_i$ of $u$ is labelled with $\beta_i$; that is, the order of the successors of $u$ is compatible with the order specified by the rule (Choice 2). An ordered, labelled tree *satisfies* $G$ if its root node is labelled with $\pi$, its frontier is $s$, and all of its nodes satisfy $G$.

*Global and local constraints* The choice of a class of reference structures for an LCG grammar formalism (Choice 1) imposes a *global* constraint on the formalism's expressivity. For example, by committing itself to ordered, labelled trees, no grammar specified in the LCG version of CFG can possibly account for syntactic structures with discontinuous configurations, and no possible choice for the constraint language (Choice 2) can change that. Similarly, in previous work [2], we have identified a class of discontinuous structures that is 'just right' for a descriptive view on Lexicalised Tree Adjoining Grammar (LTAG) [3]. Adopting this class commits an LCG formalism to subsets of those syntactic structures that are obtainable by an LTAG.

The choice of the class of reference structures is the only non-lexical constraint expressible in LCG. This sets LCG formalisms apart from other formalisms employing constraints to restrict syntactic configurations, like the ID/LP format of Generalised Phrase Structure Grammar [4] or Constraint Dependency Grammar (CDG) [5]. Both of these formalisms allow for the statement of non-lexical constraints at the level of individual *grammars* (order constraints in ID/LP grammars, all constraints in CDG). In contrast, global constraints in LCG can be imposed only by the choice of reference structures (Choice 1), which is a choice made at the level of the *formalism*. All remaining constraints are *local*: they apply to a word and the words in its immediate syntactic neighbourhood. In this sense, LCG is a *lexicalised* framework. The next section discusses the notion of locality employed in LCG and the role of lexical constraints in more detail.

*Valencies and lexical constraints* Locality is modelled through the concept of *valency*. The valency of a word $w$ specifies the possible types of a word $w$ (*accepted types*) and the number and types of other words that $w$ must connect with to form a complete expression (*required types*). The concept of valency is universal among lexicalised grammar formalisms; it is implemented by non-terminal symbols in lexicalised CFG, syntactic roles in dependency grammar, and slashed categories in categorial grammar. When we say that lexical constraints apply to words and their immediate syntactic neighbourhoods, we mean that constraints in the lexical entry for a word $w$ are relations over the words permitted by the valency of $w$. These words can be referred to by the accepted and required types of $w$.

We illustrate the idea behind lexical constraints by finalising our encoding of CFG as an LCG formalism. Assuming that we chose ordered, labelled trees as the reference class of structures (Choice 1), rules in a (lexicalised) CFG can be rewritten as LCG lexical entries using a single binary constraint relation $\prec$ to express linear precedence (Choice 2). For example, the rule $\alpha \to \beta_1 w \beta_2 \beta_3$ (where $\alpha, \beta_i \in \Pi$ and $w \in \Sigma$) would correspond to the lexical entry

$$\langle \{\alpha\}, \{\beta_1, \beta_2, \beta_3\} \,;\, \beta_1 \prec \iota \wedge \iota \prec \beta_2 \wedge \beta_2 \prec \beta_3 \rangle \,.$$

The first component of this entry specifies the types accepted by $w$, the second component specifies the required types; thus, in a tree satisfying this entry, the node labelled with $w$ must have a predecessor of type $\alpha$ and successors of types $\beta_1, \beta_2, \beta_3$. The third component of the entry contains the lexical constraints on

the valency; for the example entry, the  node labelled with $w$ (denoted by $\iota$ here) and its successors (referred to by their types) must be ordered as prescribed by the right hand side of the context-free rule. Note that this semantics exactly corresponds to McCawley's conception of CFG.

*Increasing the expressivity* Given that the LCG framework is stratified with respect to the choice of the class of reference structures and the choice of the lexical constraint languages, there are two obvious ways how the expressivity of an LCG formalism can be increased:

- choose a more permissive class of structures (for example, the LTAG structures mentioned above instead of the ordered, labelled trees employed for the encoding of CFG);
- choose other constraint languages (for example, languages with structural constraints other than precedence, like isolation or adjacency [6], or languages allowing for non-structural constraints such as agreement).

It turns out that LCG facilitates a rather detailed analysis of the implications that these two changes have in terms of the generative capacity and the processing complexity of the resulting formalisms.

One of the main reasons why one might want to experiment with expressivity alternations is that for most traditional grammar formalisms, there is a small number of 'killer phenomena' for which it seems necessary to locally extend the expressiveness of the formalisms by just the right amount. In the case of English for example, while most syntactic configurations disallow discontinuities, a few (such as in *wh*-movement) require them. It seems desireable to be able to express context-free and non-context-free phenomena in the same formalism, investing extra formal and computational resources only in cases where they really are required. We claim that LCG is suitable for such endeavours.

Another reason why we think that LCG is an interesting framework for modelling natural language is that it is able to handle linguistic phenomena that have proven to be particularly hard for other frameworks. As an example, we cite the permutation of nominal arguments in the German verb cluster known as *scrambling*. If we accept the linguistic analysis put forward by Becker et al. [7], the question whether a formalism can model scrambling boils down to asking whether it can generate the indexed language

$$\mathsf{SCR} = \{\, \pi(n^{[0]}, \ldots, n^{[k]}) v^{[0]} \cdots v^{[k]} \mid k \geq 0 \text{ and } \pi \text{ a permutation} \,\},$$

where the indices (written as superscripts) match up verbs ($v$s) with their noun arguments ($n$s). It has been shown [7] that no formalism in the class of Linear Context-Free Rewriting Systems[1] that produces a verb $v^{[i]}$ and the requirement for its matching noun argument $n^{[i]}$ in the same derivation step can generate SCR. In Section 3.3, we will present an LCG that does.

[1] The class of Linear Context-Free Rewriting Systems includes, among other formalisms, Combinatory Categorial Grammar, LTAG, and local Multi-Component TAGs.

*Structure* We start our exposition by introducing *labelled drawings* as the universal reference class of structures for LCGs (Section 2). Section 3 presents the stratified framework for constraint languages over drawings and gives some illustrative examples. In Section 4, we prove some limitative complexity results for LCG. Section 5 then addresses the issue of parsing LCGs and shows how the standard polynomial complexities for parsing can be obtained by appropriate restrictions on the structures and constraint languages. The paper concludes with an outlook on future work in Section 6.

## 2 Labelled drawings

We introduce LCGs as description languages for (labelled) *drawings* [2], a class of relational structures representing two essential syntactic dimensions: derivation structure and word order. Derivation structure captures the idea that a natural language expression can be composed of smaller expressions; word order concerns the possible linearisations of syntactic material. This section presents the basic terminology for drawings and cites some previous results.

### 2.1 Relational structures

A *relational structure* consists of a non-empty, finite set $V$ of *nodes* and a number of relations on $V$. In this paper, we are mostly concerned with binary relations on the nodes. We use the standard terminology and notations available for binary relations. In particular, $R^+$ refers to the transitive closure, $R^*$ to the reflexive-transitive closure of $R$. The notation $Ru$ stands for the relational image of $u$ under $R$: the set of all $v$ such that $(u, v) \in R$. Since relational structures with binary relations can also be seen as multigraphs, all the standard graph terminology can be applied to them.

Two types of relational structures are particularly important for the representation of syntactic configurations: *trees* and *total orders*. A relational structure $(V; \lhd)$ is a *forest* iff $\lhd$ is acyclic and every node in $V$ has an indegree of at most one. Nodes with indegree zero are called *roots*. A *tree* is a forest with exactly one root. For a node $v$, we call the set $\lhd^* v$ the *yield* of $v$. A *total order* is a relational structure $(V; \prec)$ in which $\prec$ is transitive and for all $v_1, v_2 \in V$, exactly one of the following three conditions holds: $v_1 \prec v_2$, $v_1 = v_2$, or $v_2 \prec v_1$. Given a total order, the *interval* between two nodes $v_1$ and $v_2$ is the set of all $v$ such that $v_1 \preceq v \preceq v_2$. A set is *convex* iff it is an interval. The *cover* of a set $V'$, $\mathcal{C}(V')$, is the smallest interval containing $V'$. A *gap* in a set $V'$ is a maximal, non-empty interval in $\mathcal{C}(V') - V'$; the number of gaps in $V'$ is the *gap degree* of $V'$.

### 2.2 Drawings

Drawings are forests whose nodes are totally ordered.

**Definition 1.** *A* drawing *is a relational structure* $(V; \lhd, \prec)$ *in which* $(V; \lhd)$ *forms a forest and* $(V; \prec)$ *forms a total order. If the forest structure underlying a drawing forms a tree, the drawing is called* arborescent.

Note that drawings are not the same as ordered trees: in an ordered tree, only sibling nodes are ordered; in drawings, the order is total for *all* of the nodes.

The notions of cover, gap and gap degree can be applied to nodes in a drawing by identifying a node $v$ with its yield $\triangleleft^* v$; for example, the gap degree of a node $v$ is the gap degree of $\triangleleft^* v$. The gap degree of a drawing is the maximum among the gap degrees of its nodes. We write $\mathcal{D}_g$ for the class of all drawings whose gap degree does not exceed $g$. The drawings in $\mathcal{D}_0$ are called *projective*. Fig. 1 shows three drawings of the same forest structure but with different gap degrees.

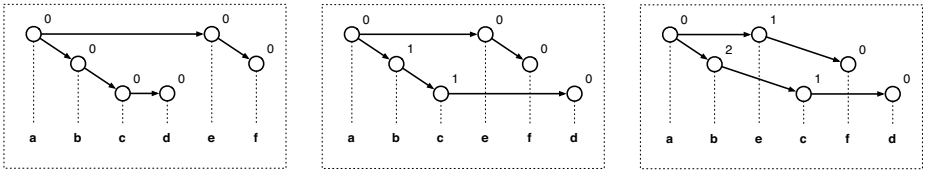



**Fig. 1.** Drawings in $\mathcal{D}_0$ (projective drawings; left), $\mathcal{D}_1 - \mathcal{D}_0$ (gap degree 1; middle) and $\mathcal{D}_2 - \mathcal{D}_1$ (gap degree 2; right). An integer at a node states that node's gap degree.

The notion of gap degree yields a scale along which the non-projectivity of a drawing can be quantified. Orthogonal to that, there are linguistically relevant *qualitative* restrictions on non-projectivity. One of these is *well-nestedness*, which constrains the possible relations between gaps [2].

**Definition 2.** *Let $\mathfrak{D}$ be a drawing. Two disjoint trees $T_1$ and $T_2$ in $\mathfrak{D}$* interleave *iff there are nodes $l_1, r_1 \in T_1$ and $l_2, r_2 \in T_2$ such that $l_1 \prec l_2 \prec r_1 \prec r_2$. The drawing $\mathfrak{D}$ is called* well-nested *iff it does not contain any interleaving trees.*

We use the notation $\mathcal{D}_{wn}$ to refer to the class of all well-nested drawings. In Fig. 1, the first and the second drawing are well-nested; the third drawing contains two pairs of interleaving trees, rooted at $b, e$ and $c, e$, respectively.

### 2.3 Labelled drawings

A labelled drawing is a drawing equipped with two total functions: one from its nodes to an alphabet $\Sigma$ of *node labels* and a second one from its edges to an alphabet $\Pi$ of *edge labels*. Since it will always be clear from the context whether we mean the node labelling or the edge labelling function, we will use the symbol $\ell$ for both: for any node $v$, $\ell(v)$ refers to the node label associated to $v$; for any edge $(u, v)$, $\ell(u, v)$ refers to the associated edge label. We write $\mathcal{D}_{\Sigma,\Pi}$ for the class of labelled drawings obtained by decorating drawings from class $\mathcal{D}$ with node labels from $\Sigma$ and edge labels from $\Pi$.

In labelled drawings, *labelled successor relations* can be defined as follows:

$$\triangleleft_\pi \;:=\; \{\, (u, v) \in V \times V \mid u \triangleleft v \text{ and } \ell(u, v) = \pi \,\}\,.$$

To reduce the complexity of our presentation, we assume the existence of a special edge label $\iota$ called 'self', distinct from all other labels, and define $\triangleleft_\iota := \mathrm{Id}$.

The *projection* of a labelled drawing $\mathfrak{D}$, $proj(\mathfrak{D})$, is the string obtained by concatenating the node labels of the drawing in the order of their corresponding nodes; this is in analogy to the notion of the frontier of an ordered labelled tree.

## 3 Lexical constraint languages

The choice of a particular class of drawings imposes a global constraint on the syntactic structures allowed by an LCG formalism. In this section, we formalise the mechanism of lexical (local) constraints. As we illustrated in the introduction, the *lexical entry* for a given word $w$ specifies the type of $w$ and the types of the words connected to $w$, and imposes additional structural restrictions using constraints from a *lexical constraint language*. In our formal model, words will correspond to node labels, and types of nodes will correspond to edge labels. A lexical constraint between two types $\pi_1, \pi_2$ in the entry of a word $\ell(u)$ will be interpreted on the nodes reachable from $u$ by the labelled successor relations named by $\pi_1$ and $\pi_2$.

### 3.1 Syntax and semantics

*Syntax* The syntax of a lexical constraint language is defined relative to an alphabet $\mathcal{R}$ of relation symbols and an alphabet $\Pi$ of edge labels. The alphabet $\mathcal{R}$, together with a function $ar$ that assigns every symbol $R \in \mathcal{R}$ a non-negative *arity* $ar(R)$, forms the *signature* of the language. We will leave the arity function implicit, and use the letter $\mathcal{R}$ to refer to signatures.

**Definition 3.** *Let $\mathcal{R}$ be a signature, and let $\Pi$ be an alphabet of edge labels. A* lexical constraint language *with signature $\mathcal{R}$ over $\Pi$, written $\mathcal{L}_\mathcal{R}(\Pi)$, consists of formulae $\phi$ of the following form:*

$$\phi \ ::= \ \mathbf{t} \mid R(\pi_1, \ldots, \pi_k) \mid \phi_1 \wedge \phi_2\,, \quad \textit{where } R \in \mathcal{R},\ ar(R) = k,\ \textit{and } \pi_i \in \Pi$$

*We write $\mathcal{L}_\mathcal{R}$ for the class of all lexical constraint languages with signature $\mathcal{R}$.*

The literal $\mathbf{t}$ is read as 'true'. We call literals of the form $R(\pi_1, \ldots, \pi_k)$ *relational constraints*. Binary relational constraints will be written using infix notation, so the notation $\pi_1\, R\, \pi_2$ will stand for $R(\pi_1, \pi_2)$.

*Semantics* The satisfaction relation associated to a lexical constraint language $\mathcal{L}_\mathcal{R}(\Pi)$ is a ternary relation between a formula $\phi$, a drawing $\mathfrak{D} \in \mathcal{D}_{\Sigma,\Pi}$ and a node $u$ in that drawing. For formulae of the form $\mathbf{t}$ and $\phi_1 \wedge \phi_2$, the definition of the satisfaction relation is the same for all lexical constraint languages:

| | |
|---|---|
| $\mathfrak{D}, u \models \mathbf{t}$ | always |
| $\mathfrak{D}, u \models \phi_1 \wedge \phi_2$ | iff $\mathfrak{D}, u \models \phi_1$ and $\mathfrak{D}, u \models \phi_2$ |

Satisfiability of relational constraints must be defined individually for a specific language. However, there are two restrictions on the possible definitions; these restrictions define lexical constraint languages in the wider sense of the term: a definition of the satisfiability relation $\mathfrak{D}, u \models R(\pi_1, \ldots, \pi_k)$ may only refer to the labelled successor relations $\{\lhd_{\pi_1}, \ldots, \lhd_{\pi_k}\}$,[2] and the question whether the defining condition applies must be decidable in time polynomial in the number of nodes in $\mathfrak{D}$. LCG does not impose any further restrictions; it allows for defining arbitrary constraint languages for labelled drawings, as long as the constraints meet the above criteria.

### 3.2 Theories and grammars

Within LCG, we distinguish between *theories* and *grammars*. Formally, an LCG *theory* is a pair of a class of (unlabelled) drawings and a class of lexical constraint languages. An LCG theory corresponds to a 'grammar formalism' in the usual sense of the word. An LCG *grammar* adopts a theory and instantiates it by choosing concrete alphabets for the node and edge labels, and a lexicon.

**Definition 4.** *Let $T = (\mathcal{D}, \mathcal{L}_\mathcal{R})$ be a theory. A* grammar *of type $T$ is a triple $G_T = (\Sigma, \Pi, Lex)$ such that $\Sigma$ is an alphabet of node labels, $\Pi$ is an alphabet of edge labels, and $Lex$ is a* lexicon *of type $\Sigma \to \mathfrak{P}(LE_\mathcal{R}(\Pi))$.*

An LCG lexicon is a mapping from node labels to sets of *lexical entries*. The type of a lexical entry depends on the signature of its constraint language and the alphabet of edge labels that the lexical constraints may refer to.

**Definition 5.** *A lexical entry describes a node in a drawing. It is a triple*

$$\langle I, \Omega\,;\phi\rangle \quad \in \quad \mathfrak{B}(\Pi) \times \mathfrak{B}(\Pi) \times \mathcal{L}_\mathcal{R}(\Pi) =: LE_\mathcal{R}(\Pi)\,,$$

*where the bags $I$ and $\Omega$ contain edge labels, and $\phi$ is a lexical constraint. A node $u$ in $\mathfrak{D} \in \mathcal{D}_{\Sigma,\Pi}$* satisfies *a lexical entry $\langle I, \Omega\,;\phi\rangle \in LE_\mathcal{R}(\Pi)$ iff*

$$\text{for all } \pi \in \Pi,\ |(\lhd_\pi)^{-1}u| = I(\pi) \text{ and } |(\lhd_\pi)u| = \Omega(\pi)\,, \quad \text{and} \quad \mathfrak{D}, u \models \phi\,.$$

The satisfaction property of a node can be lifted to the whole drawing:

**Definition 6.** *A node $u \in \mathfrak{D} \in \mathcal{D}_{\Sigma,\Pi}$* satisfies *a lexicon $Lex \in \Sigma \to \mathfrak{P}(LE_\mathcal{R}(\Pi))$ iff there is a lexical entry $\langle I, \Omega\,;\phi\rangle \in Lex(\ell(u))$ such that $u$ satisfies $\langle I, \Omega\,;\phi\rangle$. $\mathfrak{D}$* satisfies *a grammar $G$ of type $T$, written $\mathfrak{D} \models G$, iff every node $u \in \mathfrak{D}$ satisfies the lexicon of the grammar.*

### 3.3 Sample languages

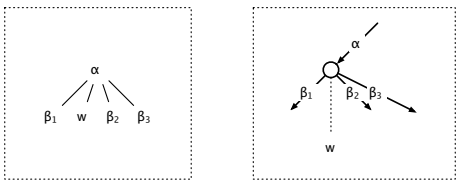
To provide an intuition for the formal concepts defined in the previous two sections, we will now translate three grammar formalisms into LCG theories. We start by adapting our previous encoding of LCFG to the new formal concepts.

[2] The definition may refer to arbitrary *unlabelled* relations in $\mathfrak{D}$.

**Lexicalised Context-Free Grammars** As already mentioned in the introduction, lexicalised context-free rules like $\alpha \to \beta_1 w \beta_2 \beta_3$ can be seen as local well-formedness conditions on node-labelled, ordered trees (see Fig. 2). To express these conditions in the formal framework defined above, we first need to choose a class of drawings suitable as models for LCFGs. Since the yields of each non-terminal are continuous, a proper choice is $\mathcal{D}_0$, the class of projective drawings. Second, we need to choose a signature for the lexical constraint language that we want to use. As we already mentioned in the introduction, the only structural constraint relevant to LCFGs is linear order. Therefore, it suffices to have a single relational constraint $\prec$ that imposes an order on the immediate successors of a node; since the language is interpreted on projective drawings, this order induces an order on the subtrees.

$$\mathfrak{D}, u \models \pi_1 \prec \pi_2 \qquad \text{iff} \quad \lhd_{\pi_1} u \times \lhd_{\pi_2} u \subseteq \prec$$

Fig. 2 shows a node-labelled tree, the corresponding lexical entry for the word $w$, and a (partial) drawing satisfying the entry. Note that (instances of) non-terminals in the LCFG rule correspond to edge labels in LCG. If $\alpha$ was a start symbol of the underlying grammar, the first component of the corresponding LCG entry would have to be the empty set; such entries can only be satisfied at root nodes.




$$w : \langle \{\alpha\}, \{\beta_1, \beta_2, \beta_3\} \,;\, \beta_1 \prec \iota \wedge \iota \prec \beta_2 \wedge \beta_2 \prec \beta_3 \rangle$$

**Fig. 2.** Encoding Lexicalised Context Free Grammars

**Lexicalised Unordered Context-Free Grammar** Since nothing forces us to impose order constraints on *all* types, we can write grammars corresponding to LCFGs with arbitrary permutations of the right hand sides of the rules. If we abandon the order constraints completely, we get the theory $(\mathcal{D}_0, \emptyset)$, which is equivalent to the class of (lexicalised) unordered context-free grammars.

**The scrambling language** The following grammar derives drawings whose projections form the scrambling language presented in the introduction. The underlying theory uses the unrestricted class of drawings and a constraint language with two literals $\prec$ (linear precedence) and $\bowtie$ (adjacency), whose semantics are

specified in Fig. 3. The grammar is $G_{\mathsf{SCR}} := (\{n, v\}, \{n, v\}, Lex)$, where the lexicon $Lex$ contains the entry $\langle\{n\}, \emptyset\,;\mathbf{t}\rangle$ for $n$ and the following entries for $v$:

$$\langle\emptyset, \{n, v\}\,; n \prec \iota \wedge \iota \prec v \wedge \iota \bowtie v\rangle\,, \quad \langle\{v\}, \{n, v\}\,; n \prec \iota \wedge \iota \prec v \wedge \iota \bowtie v\rangle\,,$$
$$\text{and} \quad \langle\{v\}, \{n\}\,; n \prec \iota\rangle\,.$$

The precedence constraints place each $v$ in between its $n$-successor and its $v$-successor. The adjacency constraint prevents material from entering between a $v$ and its $v$-successor. Therefore, all nodes labelled with $n$ must be placed to the left of all nodes labelled with $v$, and while the $v$s are ordered, the $n$s can appear in any permutation. (Fig. 3 shows a sample drawing licensed by $G_{\mathsf{SCR}}$.)

$$\begin{array}{lll} \mathfrak{D}, u \;\models\; \pi_1 \prec \pi_2 & \text{iff} & (\lhd_{\pi_1} \circ \lhd^*)u \times (\lhd_{\pi_2} \circ \lhd^*)u \subseteq \prec \\ \mathfrak{D}, u \;\models\; \pi_1 \bowtie \pi_2 & \text{iff} & (\lhd_{\pi_1} \circ \lhd^* \cup \lhd_{\pi_2} \circ \lhd^*)u \text{ is convex} \end{array}$$

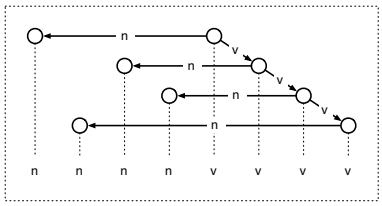



**Fig. 3.** Lexical constraint language and sample drawing for SCR

**Linear Specification Language** Suhre's LSL formalism [6] allows to generate languages with a free word order. It is inspired by ID/LP parsing [4], but allows for local constraints only, which makes it more suitable for translation into LCG. The yields in LSL are generally discontinuous; therefore, a theory for LSL needs to adopt the class of unrestricted drawings as its models. To restrict the possible linearisations, each LSL grammar rule can be annotated with local precedence and 'isolation' (zero-gap) constraints. These constraints can be translated into constraints from the lexical constraint language $\mathcal{L}_{\text{LSL}}$ shown in Fig. 4. We define the following abbreviations:

$$\diamond_\pi := \lhd_\pi \circ \lhd^*, \quad \diamond_\iota := \mathrm{Id}, \quad \diamond_\bullet := \lhd^*\,.$$

The last clause in the definition of the satisfiability relation in Fig. 4 corresponds to an isolation constraint applied to the left hand side of an LSL rule.

## 4 Limitative complexity results

The previous section has demonstrated that the LCG framework is rather expressive. This expressive power does not come without a price. It is clear that all

$$
\begin{array}{lll}
\mathfrak{D}, u \models \pi_1 < \pi_2 & \text{iff} & \diamond_{\pi_1} u \times \diamond_{\pi_2} u \subseteq \prec \\
\mathfrak{D}, u \models \pi_1 \ll \pi_2 & \text{iff} & \diamond_{\pi_1} u \times \diamond_{\pi_2} u \subseteq \prec \text{ and } \mathcal{C}(\diamond_{\pi_1} u) \cup \mathcal{C}(\diamond_{\pi_2} u) \text{ is convex} \\
\mathfrak{D}, u \models \langle \pi \rangle & \text{iff} & \diamond_{\pi} u \text{ is convex} \\
\mathfrak{D}, u \models \langle \bullet \rangle & \text{iff} & \diamond_{\bullet} u \text{ is convex}
\end{array}
$$

**Fig. 4.** Suhre's Linear Specification Language

string membership problems for LCG are in NP: we can simply guess a labelled drawing and check the lexical constraints in polynomial time. The main result of the present section is the proof that the general string membership problem for the most general LCG theory is NP-complete.

### 4.1 The general string membership problem

**Definition 7.** *Let $G = (\Sigma, \Pi, Lex)$ be a grammar for the theory $(\mathcal{D}, \mathcal{L}_{\mathcal{R}})$, and let $s$ be a string over $\Sigma$. The* general string membership problem *for $G$ and $s$, written $(G, s)$, is the problem to decide whether the following set is non-empty:*

$$\mathfrak{C}(G, s) := \{\, \mathfrak{D} \in \mathcal{D}_{\Sigma,\Pi} \mid \mathfrak{D} \models G \text{ and } \mathit{proj}(\mathfrak{D}) = s \,\}$$

*Elements of this set are called* configurations *of $(G, s)$.*

**Lemma 1.** *The general string membership problem for $(\mathcal{D}, \mathcal{L}_{\emptyset})$ is NP-hard.*

*Proof.* We will present a polynomial reduction of Hamilton Path to the general string membership problem for $(\mathcal{D}, \mathcal{L}_{\emptyset})$. More specifically, for each input graph $H = (V; E)$ to Hamilton Path, we will construct (in time linear in the size of the input graph) a grammar $G_H$ and a string $s_H$ such that $\mathfrak{C}(G_H, s_H)$ is non-empty iff $H$ has a Hamilton Path. Let $s_H$ be some string over $V$, and define

$$
\begin{array}{rcl}
\Sigma_H, \Pi_H & := & V \\
\mathit{start}(v) & := & \{\, \langle \emptyset, \{v'\} ; \mathbf{t} \rangle \mid v \to v' \in H \,\} \\
\mathit{inner}(v) & := & \{\, \langle \{v\}, \{v'\} ; \mathbf{t} \rangle \mid v \to v' \in H \,\} \\
\mathit{end}(v) & := & \{\, \langle \{v\}, \emptyset ; \mathbf{t} \rangle \mid v \to v' \in H \,\} \\
Lex_H & := & \{\, v \mapsto \mathit{start}(v) \cup \mathit{inner}(v) \cup \mathit{end}(v) \mid v \in V \,\} \\
G_H & := & (\Sigma_H, \Pi_H, Lex_H)
\end{array}
$$

Each Hamilton Path in $H$ forms a linear tree on $V$. Each such tree can be configured using $G_H$ by choosing, for each node $v$ in $H$, an entry from either $\mathit{start}(v)$, $\mathit{end}(v)$, or $\mathit{inner}(v)$, depending on the position of $v$ in the Hamilton Path. Conversely, in each configuration of $(G_H, s_H)$, each node has at most one predecessor and at most one successor qua lexicon. Therefore, each such configuration is a drawing whose successor relation forms a linear tree, and the path from the root to the leaf identifies a Hamilton Path in $H$.

To illustrate the encoding used in the proof, we show an example for an input graph $H$ and a corresponding configuration in Fig. 5. The Hamilton Path in $H$ is marked by solid edges. The depicted drawing satisfies the following lexicon $Lex_H$. (The lexical entry satisfied at each node is underlined.)

$$1 \mapsto \{\langle \emptyset, \{3\} ; \mathbf{t}\rangle, \langle \emptyset, \{4\} ; \mathbf{t}\rangle, \langle \{1\}, \{3\} ; \mathbf{t}\rangle, \underline{\langle \{1\}, \{4\} ; \mathbf{t}\rangle}, \langle \{1\}, \emptyset ; \mathbf{t}\rangle\}$$
$$2 \mapsto \{\underline{\langle \emptyset, \{1\} ; \mathbf{t}\rangle}, \langle \emptyset, \{4\} ; \mathbf{t}\rangle, \langle \{2\}, \{1\} ; \mathbf{t}\rangle, \langle \{2\}, \{4\} ; \mathbf{t}\rangle, \langle \{2\}, \emptyset ; \mathbf{t}\rangle\}$$
$$3 \mapsto \{\underline{\langle \{3\}, \emptyset ; \mathbf{t}\rangle}\}$$
$$4 \mapsto \{\langle \emptyset, \{3\} ; \mathbf{t}\rangle, \underline{\langle \{4\}, \{3\} ; \mathbf{t}\rangle}, \langle \{4\}, \emptyset ; \mathbf{t}\rangle\}$$

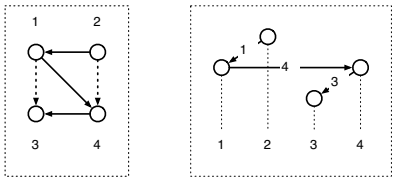



**Fig. 5.** An input graph $H$ for HAMILTON PATH and a drawing licensing $Lex_H$

### 4.2 The fixed string membership problem

The fixed string membership problem asks the same question as the general problem, but the grammar is not considered part of the input. This fact invalidates the reduction that we used in the previous section, as this reduction *constructed* a new grammar for every input, while any reduction for the fixed word problem needs to assume one fixed grammar for every input string. The proof of the following result is omitted due to space limitations:

**Lemma 2.** *The fixed membership problem for* $(\mathcal{D}, \mathcal{L}_\emptyset)$ *is polynomial.*

It would seem desirable to have a framework in which extending the signature of the constraint language may only *reduce* the complexity of the membership problem, but never increase it. For LCGs, however, this is not necessarily the case: in an unpublished manuscript, Holzer et. al. show—by a reduction of TRIPARTITE MATCHING—that for the Linear Specification Language, even the *fixed* string membership problem is NP-complete (p.c.); consequently, by the encoding of LSL presented in Section 3.3, the same result applies to LCGs.

## 5 Parsing Lexicalised Configuration Grammars

This section presents a general schema for chart-based approaches to parsing LCGs. Parsing schemata [8] provide us with a declarative specification of concrete

parsing algorithms, and allow us to analyse the complexity of these algorithms on a high level of abstraction, hiding the algorithmic details. The complexity and even the completeness heavily depend on the class of drawings that the schema is applied to. Hence we get a detailed picture of how parsers can benefit from the global constraints that are implicit in a class of drawings and up to what limits the class can be extended without losing efficiency.

### 5.1 A general parsing schema

Parsing schemata [8] view parsing algorithms as inference systems. The general parsing schema for LCG derives *parse items* representing partial drawings licensed by a given grammar and sentence. These parse items have the form $s : \langle I, \Omega \rangle$, where $s$ is a *span* (a non-empty subset of the words in the sentence) and $I$ and $\Omega$ are bags of edge labels. Each parse item represents the information that the grammar licenses a partial drawing covering the words of the input sentence specified by $s$; for this drawing to be complete, one still needs to connect its root nodes using incoming edges labelled with the labels in $I$ and outgoing edges labelled with the labels in $\Omega$. A parse item in which $\Omega$ is empty is *fully saturated*. An item $s : \langle \emptyset, \emptyset \rangle$ in which $s$ contains all the words in the sentence is *complete*.

*The lookup rule* The parsing schema contains three rules called LOOKUP, GROUP and PLUG. The LOOKUP rule creates a new parse item with a singleton span for a word $w_i$ in the input sentence:

$$\frac{\langle I, \Omega \,;\, \phi \rangle \in \mathit{Lex}(w_i)}{\{i\} : \langle I, \Omega \rangle} \text{ LOOKUP}$$

*The combination rules* The GROUP and PLUG rules derive new parse items from existing ones. The first rule, GROUP, combines two fully saturated items into a new fully saturated item. The PLUG rule saturates a bag of valencies in a parse item by combining it with another item accepting these valencies on incoming edges pointing to its root nodes:

$$\frac{s_1 : \langle I_1, \emptyset \rangle \qquad s_2 : \langle I_2, \emptyset \rangle}{s_1 \oplus s_2 : \langle I_1 \cup I_2, \emptyset \rangle} \text{ GROUP} \qquad \frac{s_1 : \langle I_1, \Omega \uplus I_2 \rangle \qquad s_2 : \langle I_2, \emptyset \rangle}{s_1 \oplus s_2 : \langle I_1, \Omega \rangle} \text{ PLUG}$$

The span of a parse item in the conclusion of the GROUP or PLUG rule ($s_1 \oplus s_2$) is the *union* of the spans in the premises ($s_1$, $s_2$). The $\oplus$ relation is a subset of the disjoint union relation. On which pairs of spans it is defined depends on the class of drawings that the schema is applied to, e.g. for $\mathcal{D}_1$ it would only be defined on pairs of spans whose union has at most one gap.

*Chart-based parsing* A concrete parsing algorithm using the general schema would test whether the inferential closure of the three rules contains a complete item. Computing the inferential closure can be done efficiently by using a *chart*, indexed

by the spans, to record parse items already derived, and by choosing a control strategy that guarantees that no two items are combined twice.

Alternatively a grammar could be translated into a definite-clause grammar (DCG): each instance of the LOOKUP rule as well as the GROUP and the PLUG rule can be represented by DCG rules. A DCG parser implemented as proposed in [9] will perform the same operations as the chart parser sketched above.

### 5.2 Completeness

Before we look at the complexity of parsing LCGs in more detail, we first need to ensure that the presented parsing schema is sound and complete, i.e., that all the inferences are valid and that every drawing can be derived with them. While this is easy to show in the general case, chart-based parsing requires a crucial invariant on the parsing rules: all spans derived during parsing must have a uniform representation. More specifically, assume that each span in the premises of a combination rule has at most $g$ gaps and thus can be represented using $2(g+1)$ integer indices (denoting the start and end positions of the $g+1$ intervals that the span consists of). Then the union of two spans must also have at most $g$ gaps. Under this side condition, the general parsing schema is no longer complete: there are drawings whose gap degree is bounded by $g$ that cannot be derived using parse items whose gap degree is bounded by $g$.

*Completeness for well-nested drawings* We will now show that for *well-nested drawings* (cf. Section 2.2), the general parsing schema *is* complete even in the presence of the gap invariant. For the proof of this result, we need the concept of the *gap forest* of a well-nested drawing [2].

**Definition 8.** *Let $(V; \lhd, \prec)$ be a well-nested drawing and let $v \in V$ be a node with $g$ gaps. The* gap forest *for $v$ is defined as the ordered forest $\mathfrak{gf}(v) = (S; \sqsupset, <)$:*

$$
\begin{aligned}
S &:= \{\{v\}, G_1(v), \ldots, G_g(v)\} \cup \{\, \lhd^* w \mid v \lhd w \,\} \\
\sqsupset &:= \textit{transitive reduction of } \{\, (s_1, s_2) \in S \times S \mid \mathcal{C}(s_1) \supset s_2 \,\} \\
< &:= \{\, (s_1, s_2) \in S \times S \mid \forall v_1 \in s_1 \colon \forall v_2 \in s_2 \colon v_1 \prec v_2 \,\}
\end{aligned}
$$

*The elements of $S$ are called* spans.

(The notation $G_i(v)$ refers to the $i$th gap in the yield of $v$.) In a gap forest, sibling spans correspond to disjoint sets whose union has at most $g$ gaps. Sibling spans belonging to the same convex region are called *span groups*.

**Lemma 3.** *Let $G$ be an* LCG *grammar and let $\mathfrak{D}$ be a well-nested arborescent drawing on nodes $V$ with gap degree at most $g$. Then $\mathfrak{D} \models G$ implies the existence of a derivation of a parse item $V : \langle I, \emptyset \rangle$ that only involves parse items whose gap degree is bounded by $g$.*

*Proof.* Let $G$ be a grammar and let $\mathfrak{D}$ be a well-nested arborescent drawing on $V$ such that $\mathfrak{D} \models G$. If $V = \{u\}$, then $\langle \emptyset, \emptyset\,;\, \phi\rangle \in \mathit{Lex}(\ell(u))$. In this case, the parse item $\{u\} : \langle \emptyset, \emptyset\rangle$ can be derived by one application of the LOOKUP rule. Now assume that $\mathfrak{D}$ consists of a root node $u$ with children $v_i$, $1 \leq i \leq k$, where each child $v_i$ is the root of an arborescent drawing $\mathfrak{D}_i$. Then

$$\langle \emptyset, P\,;\, \phi\rangle \in \mathit{Lex}(\ell(u)), \quad \text{where} \quad P = \cup_{1\leq i\leq k} \{\, \pi_i \mid \langle \{\pi_i\}, \Omega_i\,;\, \phi_i\rangle \in \mathit{Lex}(\ell(v_i))\,\}\,.$$

By induction, we may assume that each of the drawings $\mathfrak{D}_i$ was derived using parse items with gap degree at most $g$ only; in particular, each complete drawing $\mathfrak{D}_i$ corresponds to such a parse item. The drawing $\mathfrak{D}$ then can be derived using the two combination rules, successively combining the parse items for the drawings $\mathfrak{D}_i$ and the item for the root node $u$ (obtainable by the LOOKUP rule).

The interesting part of the proof is to show that the combining operations can be linearised in such a way that the gap degree of the intermediate parse items is bounded by $g$. We will now present such a linearisation, based on a post-order traversal of the gap forest for the node $u$: In a horizontal phase of the traversal, we combine all parse items corresponding to a gap group from left to right, ignoring any gap nodes. There are at most $g$ such nodes in the complete gap forest; therefore, this phase of the traversal maintains the gap invariant. In a vertical phase, we combine the parse items from the preceding horizontal phase with the item corresponding to the parent node in the gap forest in order of their gap degree. Since the gap degree of the final item is bounded by $g$, this strategy maintains the gap invariant.

### 5.3 Complexity analysis

We now determine the complexity bounds of an implementation of our schema.

*Space complexity* To bound the number of parse items stored in the chart, we look at the number of possible values for the variables of a parse item $s : \langle I, \Omega\rangle$. As both $I$ and $\Omega$ may represent arbitrary multisets over the edge labels, the number of parse items may be exponential in the size of the grammar. In the case that the drawings under consideration are unrestricted (so that a span $s$ can be an arbitrary set), the number of parse items is also exponential in the length of the input sentence. However, in cases where Lemma 3 applies, spans can be represented by $k = 2(g + 1)$ integers (cf. Section 5.2). Thus, there will be at most $O(n^k)$ different parse items in the chart.

*Time complexity* Since the chart-based architecture guarantees that no two parse items are combined twice, the space complexity can be used to bound the time complexity. Of course, if the number of parse items is exponential, the runtime of any algorithm faithfully implementing the general parsing schema will be exponential as well. In what follows, we will ignore the size of the grammar and focus on well-nested drawings with bounded gap degree. How many possibilities of combinations are there for parse items? Counted over the runtime of the complete algorithm, every parse item needs to be combined with every other item, so the time needed for these combinations is $O(n^k) \cdot O(n^k) = O(n^{2k})$.

*A refined analysis* This $O(n^{2k})$ time estimate is too pessimistic still. To see this, notice that in both of the combination rules, $k$ indices used to represent the spans only occur in the premises: since both the spans in the premises and the span in the conclusion can be represented using $k$ indices each, $2k - k$ cannot 'make it' into the conclusion. As the union operation on spans does not 'forget' about any material, the value of $k/2$ of these indices are determined by other indices in the premises. Thus, a better upper bound for the time complexity for the algorithm is $O(n^{2k-k/2})$. Remembering that $k = 2(g + 1)$, we get

**Lemma 4.** *Let $\mathcal{D}$ be a class of well-nested drawings whose gap degree is bounded by $g$, and let $\mathcal{L}_\mathcal{R}$ be a lexical constraint language. Then the general string membership problem for $(\mathcal{D}, \mathcal{L}_\mathcal{R})$ has complexity $O(2^{|G|}n^{3g+3})$.*

For context-free grammars ($g = 0$), this lemma gives the familiar $O(n^3)$ parsing result; for TAGs ($g = 1$), we get a parser that takes time $O(n^6)$. Notice that both of these complexities ignore the size of the grammar. For LCFGs, however, our parsing framework can be as efficient as e.g. the Earley parser:

**Lemma 5.** *The general string membership problem for totally ordered grammars of type $(\mathcal{D}_0, \mathcal{L}_{\{\prec\}})$ has complexity $O(|G|^2n^3)$.*

*Proof.* By the previous lemma, we know that $O(2^{|G|}n^3)$ is an upper bound. The restriction that the valency of each lexical entry are totally ordered implies that we can represent valencies as lists instead of bags.

### 5.4 The size of the grammar

The previous section offered insights in how far the model class used by a certain grammar formalism influences the completeness and the complexity with respect to the length of the input sentence. To develop an efficient parser of practical relevance based on our parsing schema however, a crucial point is the complexity with respect to the size of the grammar. Grammar size is an often neglected factor for the performance of parsing algorithms: a standard sentence of, say, 25 words, is usually several orders of magnitude shorter than a lexicalised grammar. While grammar size thus is significant even for frameworks in which the grammar only contributes linearly or quadratically to the speed of the parsing algorithm (such as context-free grammar), it is definitely an issue in a framework like LCG, where for reasons of expressive power it cannot in general be avoided. It seems then, that it is desirable to complement the chart-based parsing architecture by methods to avoid the worst-case complexity in the size of the grammar whenever possible. This is where we propose to use constraint propagation: lexical constraints can be used to control the chart-based parser. To give a very simple example: in the presence of order constraints, far from all of the possible combinations of parse items need to be considered when applying the PLUG rule: if an item $i$ has open valencies $\pi_1 \prec \pi_2$, there is no need to try to plug $\pi_2$ with an item adjacent to $i$—any item plugging $\pi_1$ precedes any item plugging $\pi_2$ in all licensing drawings. How exactly the interaction between constraint propagation and chart parsing it realized and how much a parser can benefit from each single constraint are open questions that we are currently addressing.

## 6 Conclusion

This paper presented Lexicalised Configuration Grammars (LCGs), a novel framework for the descriptive analysis of natural language. LCG is stratified with respect to two parameters: the choice of a class of reference structures (a global constraint), and the choice of a lexical (i.e., local) constraint language used to describe those structures that should be considered grammatical. Translating grammar formalisms into LCG makes it possible to study these formalisms and their relations from a new perspective, and to experiment with gradual and local alternations of their expressivity and processing complexity. LCGs are expressive enough to generate the scrambling language, a language that cannot be generated by many traditional generative frameworks. The general string membership problem for LCG is NP-complete; however, a broad class of linguistically relevant LCGs can be parsed in polynomial time.

*Future work* We plan to continue our research by investigating the potential of the processing framework outlined in Section 5 to combine chart-based and constraint-based processing techniques. Our immediate goal is the implementation of a parser for LCGs that uses constraint propagation to avoid the worst-case complexity of the chart-based parsing algorithm with respect of the size of the grammar. One of the major technical challenges in this is the constraint-based treatment of lexical ambiguity: handling disjunctive information is notorously difficult for constraint propagation. In a second line of work, we will try to relate LCGs to more and more traditional grammar formalisms by defining appropriate LCG theories and grammars and proving the necessary equivalence results.

## References


1. McCawley, J.D.: Concerning the base component of a transformational grammar. Foundations of Language **4** (1968) 243–269
2. Bodirsky, M., Kuhlmann, M., Möhl, M.: Well-nested drawings as models of syntactic structure. In: 10th Conference on Formal Grammar and 9th Meeting on Mathematics of Language, Edinburgh, Scotland, UK (2005)
3. Joshi, A., Schabes, Y.: Tree Adjoining Grammars. In: Handbook of Formal Languages. Volume 3. Springer (1997) 69–123
4. Gazdar, G., Klein, E., Pullum, G.K., Sag, I.A.: Generalized Phrase Structure Grammar. Havard University Press, Cambrige, MA (1985)
5. Maruyama, H.: Structural disambiguation with constraint propagation. In: 28th Annual Meeting of the Association for Computational Linguistics (ACL 1990), Pittsburgh, Pennsylvania, USA (1990) 31–38
6. Suhre, O.: Computational aspects of a grammar formalism for languages with freer word order. Diploma thesis, Universität Tübingen (1999)
7. Becker, T., Rambow, O., Niv, M.: The derivational generative power, or, scrambling is beyond lcfrs. Technical Report IRCS-92-38, University of Pennsylvania (1992)
8. Sikkel, K.: Parsing Schemata: A Framework for Specification and Analysis of Parsing Algorithms. Springer-Verlag (1997)
9. Shieber, S.M., Schabes, Y., Pereira, F.C.N.: Principles and implementation of deductive parsing. Journal of Logic Programming **24** (1995) 3–36

# N:M Mapping in XDG –The Case for Upgrading Groups

Jorge Marques Pelizzoni and Maria das Graças Volpe Nunes

Universidade de São Paulo, Instituto de Ciências Matemáticas e de Computação
Av. do Trabalhador São-Carlense, 400. CEP 13560-970. São Carlos – SP – Brasil
{jorgemp, gracan}@icmc.usp.br
http://www.nilc.icmc.usp.br

**Abstract.** The eXtensible Dependency Grammar (XDG) is a very promising CP-natural framework with which to tackle varied NLP problems and their combinatorial complexity. XDG draws heavily on its non-transformational character for efficiency, which opens the issue of N:M mapping e.g. between syntactic and semantic structures. We resume discussion on this issue and attempt to demonstrate that improvement of the available solutions is at once desirable and crucial. To this end, we assess their suitability in several scenarios assuming a syntax-semantics interface: parsing *vs.* generation; treating Multiword Expressions; treating connectives introducing optional components such as adverbials; etc. Finally, we propose some guidelines to overcome the identified limitations.

## 1 Introduction

The intuition has been widespread for some time so far that language processing – whether natural or artificial – emerges from the interplay of various concurrent constraints operating at or between different levels of analysis. At times it seems almost possible to feel determinacy ebbing and flowing through such constraint systems e.g. as the potential ambiguity of the words in a sentence gradually reduces until satisfactory interpretations become available. This is especially true of natural processors, i.e. humans, when tackling a second language. As the paradigms of *Constraint Satisfaction* (CS) and particularly *Constraint Programming* (CP) have arisen in Computer Science exactly to mimic this ebb-and-flow intuition suitable to tame complexity in a whole range of problems, no wonder language processing is also in focus.

However, CP faces therein a twofold challenge: not only does the (in)exact nature of linguistic constraints and objects still elude us all, CP practitioners or not, but also much of the linguistic tradition, drawing heavily upon transformational primitives, is not usually amenable to straightforward or *efficient* modelling, to say the least. Therefore, the latest years have seen much effort to strengthen propagation in grammar modelling, which often led to alternative constraint-based frameworks. The move from early (and already not so mainstream) Tree Adjoining Grammar-based frameworks [10,11,12] to more recent Dependency Grammar-based ones [9,8,7] is

certainly bound for stronger propagation, regardless and arguably to the detriment of explanatory adequacy.

The *eXtensible Dependency Grammar* (XDG) [4,5,6] is perhaps the latest stage in this evolutionary line and represents an important leap from its ancestors. Even though still leaving much room for further development as we shall presently see, XDG achieves an unprecedented balance between (i) complexity, which is hopefully controlled by the strong propagation it inherits from DG-based frameworks, (ii) generality, i.e. potential coverage of phenomena or application spectrum, which is significantly broadened by XDG's underspecification, extensibility and novel *multidimensional* metaphor, and (iii) instantiability, i.e. ease of instantiation or application, which benefits from XDG's enhanced support for modularity and reusability. As a generality bonus, XDG is such a CP natural that a CP implementation can actually achieve *bidirectionality*, i.e. the property that one same grammar might be used both for analysis and generation with the same search engine modulo I/O processing.

In that respect, Debusmann et al. [5] have already sketched a XDG-based relational (i.e. bidirectional) syntax-semantics interface. However, they bypassed the issue of **N:M mapping** – or **subgraph handling** – then, which is nonetheless unavoidable if one is ever to tackle most function words, especially connectives and auxiliary verbs, support verbs (such as "do" in "do the dishes" or "have" in "have an argument with") and multiword expressions (MWEs), since these usually involve worthy syntactic nodes that in spite of influencing interpretation have no semantic counterpart. The issue has been addressed by Debusmann elsewhere [2] specifically with a focus on MWEs, which were tackled in XDG by means of a restricted form of grouping and deletion, herein referred to simply as the *group* construct. This remains the sole such account so far and, ingenious as it is to provide a very promising technique, it does not go beyond MWEs and leaves, even in that matter, several open issues. The general purpose of this paper is exactly to resume discussion from that point and address some of these issues. Rather than providing definitive solutions, our highest goal is to demonstrate that improvement is at once desirable and crucial while gathering requirements for future developments.

First of all, we provide a little background on XDG (Section 2) and review Debusmann's group technique (Section 3). Next we demonstrate some of its shortcomings (Section 4), as (i) when treating connectives introducing optional components, like adverbials, (ii) when some particularities of MWEs come into play, and (iii) when one consider the convenience of an $N_1{:}N_2{:}\ldots{:}N_n$ mapping generalization. Having presented this rationale and thus made the main point of the paper, we very briefly provide some pointers as to future work on the lexicon component towards overcoming those limitations (Section 5). At all times our main concerns are those of generality (i.e. broaden the coverage of groups, also ensuring bidirectionality), instantiability (make groups usable), and complexity/scalability (make groups feasible in terms of lexicon storage and keep propagation strong).

## 2 XDG Background

We start by reviewing Debusmann's treatment of MWEs [2] by means of groups, which requires acquaintance with XDG's core concepts. Therefore, an informal overview of these concepts is also in order. For a formal description of XDG, however, see Debusmann et al. [4,5,6].

Most of XDG's strengths stem from its *multidimensional* metaphor (see Fig. 1), whereby an (holistic or multidimensional) XDG analysis consists of a set of concurrent, synchronized, complementary, mutually constraining one-dimensional analyses, each of which is itself a *graph* sharing the same set of vertices as the other analyses, but having its own type or *dimension*, i.e., its own edge label and lexical feature types and its own well-formedness constraints. In other words, each 1D analysis has a nature and interpretation of its own, associates each vertex with one respective instance of a data type of its own (lexical features) and establishes its own relations/edges between vertices using labels and principles of its own. For example, an XDG grammar/analysis might have four dimensions/1D analyses, two for syntax (immediate dominance and linear precedence) and two for semantics (predicate argument structure and scope), as in Debusmann et al. [5].

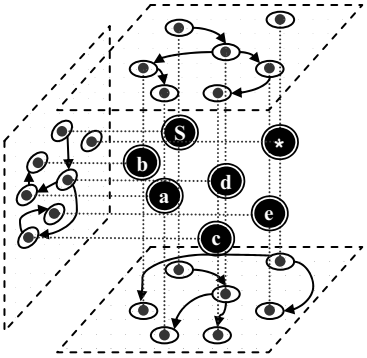



**Fig. 1.** Three concurrent one-dimensional analyses. It is the sharing of one same set of vertices that co-relates and synchronizes them into one holistic XDG analysis.

That might sound rather autistic at first, but the 1D components of an XDG analysis interact in fact. It is exactly their sharing one same set of vertices, whose sole intrinsic property is identity, that provides the substratum for interdimensional communication, or rather, *mutual constraining*. That is chiefly achieved by means of two devices, namely: interdimensional principles and lexical synchronization.

**Interdimensional Principles.** Principles are reusable, usually parametric constraint predicates used to define grammars and their dimensions. Those posing constraints between two or more 1D analyses are said *interdimensional*. For example, one XDG grammar defining one dimension to capture predicate argument (PA, modelling semantic roles) structure and another one for immediate dominance (ID, modelling syntactic relations) may prescribe a principle between them ensuring that, for every vertex *v* fulfilling some lexical precondition, whenever there is an edge *pat(ient)* from

*v* to some other vertex *w* on dimension PA, then there is an edge *subj(ect)* to *w* on dimension ID. This constraint is more formally expressed thus:

$$\forall v \in V\left( unaccusative(v) \Rightarrow \forall w \in V\left( v \underset{PA}{\overset{pat}{\rightarrow}} w \Rightarrow \exists u \in V\left( u \underset{ID}{\overset{subj}{\rightarrow}} w \right) \right) \right), \quad \textbf{(1)}$$

where *V* is the set of vertices of the current analysis.

This example principle is subsumed by the parametric principle *linkingEnd* available in the XDG Development Kit[1] (XDK). Each application of *linkingEnd* takes three parameters, namely two dimensions $D_1$ and $D_2$ and one function

$$linkf: V \times V \times label(D_1) \rightarrow 2^{label(D_2)}, \quad \textbf{(2)}$$

where *label(D)* denotes de set of all possible edge labels on dimension *D* and whose function is thus map edges *(v,w,l)* on $D_1$ into sets of edge labels on $D_2$. The meaning of *linkingEnd*($D_1$,$D_2$,*linkf*) can be more formally expressed thus:

$$\forall v, w, l\left( v \underset{D_1}{\overset{l}{\rightarrow}} w \Rightarrow linkf(v,w,l) = \varnothing \vee \exists u, l'\left( l' \in linkf(v,w,l) \wedge u \underset{D_2}{\overset{l'}{\rightarrow}} w \right) \right). \quad \textbf{(3)}$$

Parameter *linkf* may well be set by an application of

$$defaultf(D) = \lambda(v,w,l).\ lex_D(v)('end')(l), \quad \textbf{(4)}$$

where $lex_D(v)$ maps vertex *v* into a function giving access to its lexical features according to dimension *D*. Feature *end*, in turn, should denote a function mapping $D_1$ edge labels into subsets of $label(D_2)$. Therefore, the constraint in Equation (**1**) can easily be implemented by *linkingEnd(PA,ID,defaultf(PA))*, provided that unaccusative verbs have lexicon entries whose *end* features for dimension *PA* map label *pat* into the singleton $\{subj\}$.

Some remarks are worth making as regards the above example, namely (i) that shared vertices constitute the real and only meeting points between 1D analyses, (ii) that interdimensional principles strongly rely on (i) to do their job, and (iii) that most of them are *lexicalized*, i.e. impose constraints that depend on the lexical features of vertices according to the dimensions involved. Further details on lexicalization are provided below.

**Lexical Synchronization.** As pointed out above, principles, whether intra- or interdimensional, usually resort to the lexical features of vertices. This implies that any XDG instance has a lexicon as a component, which is specified in two steps: first, each dimension declares its own lexicon entry type, i.e. an Attribute-Value Matrix (AVM) type; next, once all dimensions have been declared, lexicon entries are provided, each specifying the values for features on all dimensions. Finally, at runtime it is required of well-formed analyses that there is at least one valid assignment of lexicon entries to vertices such that all principles are satisfied. In other words, every vertex must be assigned a lexicon entry that *simultaneously* satisfies all

[1] http://www.ps.uni-sb.de/~rade/xdg.html.

principles on all dimensions, for which reason the lexicon is said to *synchronize* all 1D components of an XDG analysis.

Lexical synchronization is a major source of propagation. Resuming our unaccusative verb example and assuming a text generation scenario, if it happens to be known at some point that there are no more sources of *subject* edges available on the ID dimension, then lexicon entries corresponding to unaccusative realizations of an as yet unrealized verb shall be discarded. That might further narrow the domains of variables on various dimensions and trigger further propagation.

## 3 The State of the Art – Groups

It must be clear from the previous section that the sharing of one same set of vertices by the 1D components of an XDG analysis is key to the framework and that, formally speaking, deletion or insertion operations would be out of the question. On the face of it, that might seem lethal to any higher aspirations on the part of an XDG-based syntax-semantics interface. Indeed, for starters, it is reasonable to expect that MWEs such as "(to) have an argument with" and "argue with" should correspond to one same single semantic literal, say *argue*, in spite of comprising multiple vertices on syntactic dimensions (one per word). On the other hand, it is sometimes the case that a semantic subgraph should correspond to fewer syntactic nodes, as in instrument incorporation. For example, "cut with a knife" is likely to have a semantic representation comprising at least two semantic vertices, whose realization in some languages, however, would take one single word (for one, the Brazilian Sign Language). Finally, in general, one would not expect vertices corresponding to connectives (prepositions and conjunctions) to have direct counterparts on a semantic dimension, much though they should be taken into account during interpretation or somehow produced during generation.

**Emulating Deletion.** Debusmann [**2**] introduces a simple though clever technique to circumvent this limitation, or rather, to carry out N:M mapping on top of XDG. The basic idea behind his technique is emulating deletion thus: whenever a vertex has but one incoming edge with a reserved label, say *del*, it is considered as virtually deleted. In addition, one artificial "root" node is postulated from which emerge as many *del* edges as required on all dimensions. Given that lexicon entries usually rule the valency of vertices on each dimension separately, i.e. state which labels they accept on incoming and outgoing edges, it is straightforward to state that a vertex should be deleted on certain dimensions, it sufficing to provide lexicon entries accepting only and necessarily one incoming *del* edge on the referred dimensions.

Crucial though it is, the possibility of deletion by itself is not enough. There must also be a way to treat certain subgraphs as units, or rather *groups*, whose rationale must be given from two complementary points of view.

**The Group Coherence Problem.** From the point of view of parsing, the fact must be captured that the group meanings of "have a word with" or "take out" compete with other readings of their component words that are applicable in other contexts ("out" can even be an adjective!). In other words, treating "have a word with" as a block in

XDG implies that *each* word will have, among its lexicon entries, one *have-a-word-with* entry, i.e. one exclusively reserved to the group reading. Now, when parsing "write down a word", the *have-a-word-with* entry of "word" will compete with the applicable entry and should be guaranteed to lose. What is worse, any *have-a-word-with* entry should be guaranteed to lose unless enough *have-a-word-with* entries (one for each of the component words) are available at the same time for the sentence at hand.

That problem will herein be referred to as the **Group Coherence Problem (GCP)**. Further generalizing and introducing some useful terminology: (i) every group reading is built by selecting a set of **grouped lexicon entries**; (ii) any such set is said to be a **group instance**; (iii) in order for a grouped lexicon entry to be selected it is necessary that enough **cogrouping entries**, or rather, enough of its **cogroupers** are available; (iv) given two group instances $G_1$ and $G_2$, they are said to be **(group) alternates** iff there exist two entries $e_1 \in G_1$ and $e_2 \in G_2$ such that $e_1$ and $e_2$ cogroup (e.g. "take out" and "takes out" may well be parsed by group alternates); (iv) the **paradigm** of a group instance $G$ is the union of all its alternates according to a lexicon (e.g. the set of all lexicon entries necessary to parse "take/takes/took/taking/etc. out").

The GCP can thus be rephrased as the problem of identifying well-formed group instances and ensuring that all cogroupers are simultaneously in the same state, either selected or discarded. The solution proposed by Debusmann is based again on a simple though ingenious idea. It consists of (i) capturing the inner structure of group instances and (ii) requiring it to be wholly reconstituted by cogrouping entries only. If (ii) cannot be satisfied, it means that there is at least one necessary cogrouper missing, no group instance can be built, and all involved cogroupers are discarded.

The trick is not so easily implemented now. First of all, every group paradigm $P$ is assigned one unique identifier *gid(P)*. Next, every lexicon entry must bear its group id, even non-grouped entries (which might all take one same reserved null id). To this end, one dimension must specify a special lexical feature, say *group*, to hold this bit of information. Finally, all dimensions bearing structure (some may axiomatically hold unconnected vertices only) must specify a lexical feature, say *outgroups*, of the type

$$lex_D(v)('outgroups'):label(D)\rightarrow 2^{dom(gid)}, \qquad (5)$$

where *dom(f)* denotes the domain of function $f$. For every dimension $D$ having this feature, *outgroups* maps edge labels into sets of group ids and is intended to shortlist the otherwise worthy receivers of the edges emerging from any given vertex. More specifically, given a pair of vertices *src* and *target* whose respective valencies otherwise sanction an edge *(src,target,label)* on dimension $D$, any such edge will be nonetheless inhibited unless the group of (the selected lexicon entry for) *target* belongs to $lex_D(src)('outgroups')(label)$. This new constraint is the **group coherence principle** and must hold for every structure-bearing dimension.

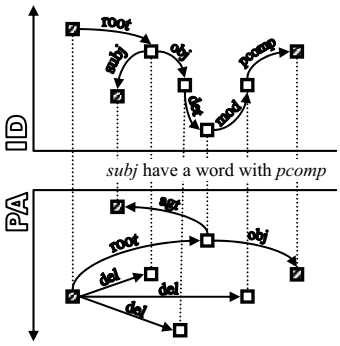



**Fig. 2:** analisys of an occurrence of the MWE "have a word with" showing dimensions ID (syntactic) and PA (semantic, in which there is deletion). Shaded vertices are external to the group at issue

For example, supposing that "have a word with" should have the structure depicted in Fig. 2 and correspond to one single semantic literal *talk*, the rows in Table 1 show what the respective lexicon entries would look like for a selection of relevant features. In this table, wherever relevant, features are subscripted with their respective dimensions. For example, the forth column regards feature *outgroups* on dimension ID. Apart from the already introduced features or otherwise self-explanatory ones, *in* resp. *out* compose the valency of a vertex, i.e. hold information regarding (i) which labels are accepted for incoming resp. outgoing edges and, for each listed label *l*, (ii) a cardinality constraint, stating e.g. whether at least one *l* edge is required (*l* or *l*+) or not (*l?* or *l**) and whether the vertex may accept at most one *l* edge (*l* or *l?*) or more (*l** or *l*+). On the second column, *(del)* is but a special literal reserved for deleted vertices.

**Table 1.** Lexical entries encoding the grouping of "talk to" and "have a word with"

| **word** | **literal** | **group** | **outgroups$_{ID}$** | **in$_{ID}$** | **out$_{ID}$** | **in$_{PA}$** | **out$_{PA}$** | **link$_{PA}$** |
|---|---|---|---|---|---|---|---|---|
| talk | *talk* | g1 | $\{iobj \mapsto \{g1\}\}$ | $\{root\}$ | $\{subj, obj\}$ | $\{root\}$ | $\{agt, obj\}$ | $\begin{Bmatrix} agt \mapsto \{subj\} \\ obj \mapsto \{obj\} \end{Bmatrix}$ |
| to | (del) | g1 | $\varnothing$ | $\{iobj\}$ | $\{pcomp\}$ | $\{del\}$ | $\varnothing$ | $\varnothing$ |
| have | (del) | g2 | $\{obj \mapsto \{g2\}\}$ | $\{root\}$ | $\{subj, obj\}$ | $\{del\}$ | $\varnothing$ | $\varnothing$ |
| a | (del) | g2 | $\{det \mapsto \{g2\}\}$ | $\{obj\}$ | $\{det\}$ | $\{del\}$ | $\varnothing$ | $\varnothing$ |
| word | *talk* | g2 | $\{mod \mapsto \{g2\}\}$ | $\{det\}$ | $\{mod\}$ | $\{root\}$ | $\{agt, obj\}$ | $\begin{Bmatrix} agt \mapsto \{subj\} \\ obj \mapsto \{obj\} \end{Bmatrix}$ |
| with | (del) | g2 | $\varnothing$ | $\{mod\}$ | $\{pcomp\}$ | $\{del\}$ | $\varnothing$ | $\varnothing$ |

**Vertex Expansion.** By reconsidering the same "have a word with" example now from the point of view of generation (i.e. given input on dimension PA, try to reconstruct valid analyses on dimension ID), an additional issue emerges closely related to that of model creation in CP and named herein the *Vertex Expansion Problem* (VEP). As far as regards actual input to a generation system, one might well expect to generate e.g. "They have a word with Mary" from a simple three-vertex PA

graph as seen in Fig. 3, which obviously falls short of vertices. It should be clear by now that, under grouping, there is more to setting up the XDG scene – or rather, model creation – than simply transferring the input graph onto the PA dimension and naïvely looking literals up on the lexicon. A *group-oriented* lookup procedure is strictly needed, even though missing in XDG's current implementation.

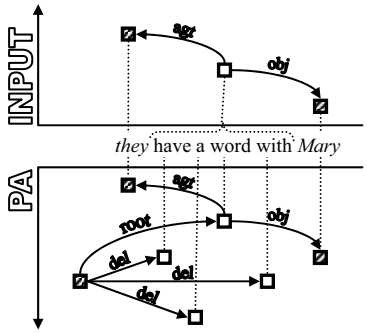



**Fig. 3:** an instance of the Expansion Problem in generation. Although PA is the input dimension, the actual input lacks vertices

Debusmann has already pointed out the need for such a procedure and proposed one. First, he states that, for generation, a function

$$groups : Sem \rightarrow 2^{dom(gid)} \tag{6}$$

(where *Sem* is the set of all semantic literals) is needed mapping any given literal *lit* to a set containing the ids of all groups realizing *lit*. During model creation, for each literal *lit*: (i) the set *Ps* is retrieved of the paradigms[2] of all groups in *groups(lit)*; (ii) then the set *V* is created of new vertices such that $|V| = \max\{|P| : P \in Ps\}$; (iii) for each paradigm $P \in Ps$, its entries are arbitrarily assigned to nodes in *V*, i.e., an arbitrary injection $assign_P : P \rightarrow V$ is constructed; then (iv) the base set of alternative lexicon entries for each vertex $v \in V$ is constructed thus:

$$entries(v) = \bigcup_{P \in Ps} \begin{cases} assign_P^{-1}(v), & v \in dom\left(assign_P^{-1}\right); \\ \varnothing, & otherwise \end{cases} \tag{7}$$

(v) finally, the *actual* set of alternative lexicon entries for each vertex is defined thus:

$$entries'(v) = \begin{cases} entries(v), & |entries(v)| = |V| \\ entries(v) \cup Del, & otherwise \end{cases}, \tag{8}$$

where *Del* is a constant lexical entry allowing simultaneous deletion on all dimensions and thus accounting for paradigms requiring fewer nodes than available.

For an example application of this method, one might consider the realization of literal *talk* according to the lexicon in Table 1. The number of vertices created for this

[2] Specializing the general definition for Debusmann's solution, a set of lexicon entries sharing one same group id is a group paradigm iff it contains all lexicon entries sharing that id.

sole literal would be four, say $\{u, w, v, x\}$, so as to hold the paradigm with the greatest number of entries ("have a word with"). One valid actual assignment of alternative lexical entries is given in Table 2, which states, for example, that vertex *u* will be selecting either the first or the fifth row/entry. Notice that vertices *v* and *x* will select the special *Del* entry in the event of "talk to" being generated (third and forth rows).

**Table 2.** During generation, actual lexical entries allowing selection between two alternative realizations ("talk to" and "have a word with") of one same semantic literal *talk*. The first column coindexes entries competing for one same vertex

| assigned to vertex | word | lit. | group | $\text{outgroups}_{\text{ID}}$ | $\text{in}_{\text{ID}}$ | $\text{out}_{\text{ID}}$ | $\text{in}_{\text{PA}}$ | $\text{out}_{\text{PA}}$ | $\text{link}_{\text{PA}}$ |
|---|---|---|---|---|---|---|---|---|---|
| **u** | talk | *talk* | g1 | $\{iobj \mapsto \{g1\}\}$ | $\{root\}$ | $\{subj, obj\}$ | $\{root\}$ | $\{agt, obj\}$ | $\begin{Bmatrix} agt \mapsto \{subj\} \\ obj \mapsto \{obj\} \end{Bmatrix}$ |
| **w** | to | (del) | g1 | $\varnothing$ | $\{iobj\}$ | $\{pcomp\}$ | $\{del\}$ | $\varnothing$ | $\varnothing$ |
| **v** | (del) | (del) | null | $\varnothing$ | $\{del\}$ | $\varnothing$ | $\{del\}$ | $\varnothing$ | $\varnothing$ |
| **x** | (del) | (del) | null | $\varnothing$ | $\{del\}$ | $\varnothing$ | $\{del\}$ | $\varnothing$ | $\varnothing$ |
| **u** | have | (del) | g2 | $\{obj \mapsto \{g2\}\}$ | $\{root\}$ | $\{subj, obj\}$ | $\{del\}$ | $\varnothing$ | $\varnothing$ |
| **w** | a | (del) | g2 | $\{det \mapsto \{g2\}\}$ | $\{obj\}$ | $\{det\}$ | $\{del\}$ | $\varnothing$ | $\varnothing$ |
| **v** | word | *talk* | g2 | $\{mod \mapsto \{g2\}\}$ | $\{det\}$ | $\{mod\}$ | $\{root\}$ | $\{agt, obj\}$ | $\begin{Bmatrix} agt \mapsto \{subj\} \\ obj \mapsto \{obj\} \end{Bmatrix}$ |
| **x** | with | (del) | g2 | $\varnothing$ | $\{mod\}$ | $\{pcomp\}$ | $\{del\}$ | $\varnothing$ | $\varnothing$ |

## 4 Shortcomings and Requirements Gathering

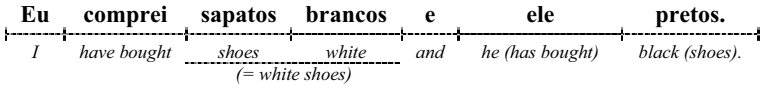
Now we are in a position to sense the limits of the solutions presented in the previous section and thus gather requirements for future enhancements to XDG and its implementation. Our main goal in this section is provide evidence pointing towards a new balance between maybe the most abstract requirements on frameworks, namely the trinity (i) generality, i.e. coverage/expressibility of phenomena, (ii) instantiability, i.e. ease of instantiation, and (iii) complexity. Instantiability is the design-time analogue of complexity, a runtime concept. In fact, even if some methodology is theoretically applicable with satisfactory accuracy (i.e. it is general enough), it is likely to be discarded if its application happens to be too costly. In other words, it is practicality, feasibility, ease, reasonable demand in resources, in summary, instantiability during development and complexity during execution that will eventually drive developers' preference for this or that framework. As regards XDG, instantiability concerns features and primitives of the grammar specification language, while complexity is related to model creation, propagation and stored lexicon size.

### 4.1 Generality *vs.* Expansion – Bidirectionality and Null Categories

Debusmann [2] has focused on MWEs only and implied that group-oriented lexicon lookup would be a generation trait; however, vertex expansion is not restricted to

generation, which is already suggestive that such an enhanced lookup/model creation procedure is actually the general – rather than the exceptional – case. Take for instance instrument incorporation in the whole set of "cut-with-a-X" words in Brazilian Sign Language [1], not to mention manner and intensity incorporation.

Although work is lacking on this specific matter, if ellipsis is ever to be handled in XDG, that will probably resort to some kind of vertex expansion during parsing. Consider, for example, the following Portuguese sentence[3]:




However such a hard phenomenon is to be tackled, one can count on the fact that "Eu" and "ele" cannot share the same vertices for "(has/have) bought" or "shoes", at least not directly. Generally speaking, all evidence suggests that, if **syntactic null categories** are ever to be "parsed", that will require vertex expansion as they cannot rely on the presence of words of their own to trigger the generation of their respective vertices during model creation.

### 4.2 Complexity *vs.* Expansion

**Inflecting MWEs.** One issue that Debusmann has left open in his XDG account of MWEs is related to inflection, i.e. how such alternations as "have/has/had an/∅ argument/s[4] with" are to be encoded. As we shall presently see, the current solution might have undesired side-effects on complexity, either on lexicon storage or propagation. In the discussion below, one should always take into account that there are languages considerably more inflected than English. Romance languages, for example, deliver tenths of distinct inflected forms for every single verb, counting out the so-called *compound forms*, i.e. those involving auxiliary verbs.

The obvious first impulse would be rather flawed to encode a whole grammatical paradigm into one sole group paradigm, i.e. to make all inflections of a given base MWE share the same group id, *which might even work for parsing* (strictly without vertex expansion) but not generation[5]. Take for instance the grammatical paradigm "have/has/had a word with". In addition to the relevant entries in Table 1, its encoding would also require two further entries – for "has" and "had" respectively – quite similar to that of "have". Supposing they all share the same group id and assuming a generation scenario, on application of the vertex expansion procedure described in the previous section six vertices would be created instead of the correct four, and either failure or malformed output would follow. Apropos, in generating a Romance language over forty vertices would usually sprout. This so to speak vertex prodigality stems from the fact that, in this misencoding, there is no clue whatsoever that some entries should compete for one same vertex.

---

[3] Thanks to Denys Duchier, originally in German.

[4] The alternation "had an argument with" *vs.* "has arguments with" can arguably be regarded as involving aspect inflection.

[5] One corollary worth deriving from the following facts is that XDG is only *potentially* bidirectional, i.e. there are grammars that might work in one direction but not in the other.

Restricting ourselves to valid inflection schemes, we were able to devise two such designs, both of which cannot help creating one exclusive group paradigm for each inflection. They differ, however, in the possibility of paradigms sharing some entries. Unfortunately, both of them entail complexity side-effects as explained below.

**No Sharing: Storage Complexity and Overactivation.** Let us first assume the simplest solution, namely create as many group paradigms as there are inflections ensuring that they do not share one entry whatsoever. In this design, if "have a word with" takes four lexicon entries, then so does "has a word with", and altogether this makes eight distinct entries. The sole difference between the whole lot of "a/word/with" entries lies in their group ids.

The design works but has two major disadvantages, namely: (i) storage requirements are subject to a *significant multiplying factor*, which will get much worse by the end of this section; (ii) this factor also affects model creation inasmuch as **overactivation** is likely to occur, i.e. having vertices select from loads of virtually equivalent lexicon entries. For example, in Portuguese, either when generating or parsing something like "ter uma discussão com" ("have an argument with"), there would be three vertices trying to select from over forty different entries for "discussão", "uma" and "com" respectively, which, ironically, are invariable in this MWE.

**Sharing: All the Same or Spurious Symmetries.** An alternate encoding method would be keep one group paradigm per inflection sharing equivalent entries (modulo group ids) with all the others. Enabling sharing would require a minor adaptation to Debusmann's original solution, namely (i) replace feature *group* (the id of the group to which an lexical entry belongs) with say *groups* (a set thereof), (ii) restate the Group Coherence Principle to hold for every candidate edge *(src,target,label)* on every structure-bearing dimension *D* thus:

$$lex_{GD}(target)('groups') \cap lex_{D}(src)('outgroups')(label) \neq \varnothing, \qquad \textbf{(9)}$$

where *GD* (a constant) is the dimension holding feature *groups*.

For example, in this design, "a/word/with" would take one single entry in the encoding of the whole "have a word with" grammatical paradigm. However, this solution comes in two radically different flavours depending on whether group paradigms originating from different grammatical paradigms may be sharers. In ground terms, depending on whether the entries for "a" and "with" might be shared by all inflections not only of "have a word with" but also of "have a fight/quarrel/argument/etc. with" or, more relevantly still, whether entries for the inflections of "have" might also be shared by all groups paradigms in which "have" acts as a support verb. Let us call **restrained** resp. **unrestrained sharing** the solution obtained by refusing resp. accepting those conditions.

Unrestrained sharing certainly answers both the storage complexity and overactivation problems posed by no sharing at all. However, it incurs propagation loss, especially in generation, when the order of vertices is not predetermined, which *might* otherwise help disambiguation. In either direction, odds are that propagation only will not be able to reconstitute the internal structure of groups, as various vertices may potentially belong to one given group, though not *simultaneously*, only *alternately*, which is known to kill propagation. In other words, **symmetries** are likely

to be introduced by the encoding scheme – and **spurious** at that, inasmuch as not natural of the problem at hand, but rather brought in by the adopted solution. For example, consider the generation of the following sentence:

I had a word with the director after having a quarrel with one of my students.
$MWE_1$ $MWE_2$

Under unrestrained sharing, vertex expansion would yield two "a/with" vertices, respectively for $MWE_1$ and $MWE_2$. However and precisely due to sharing, both vertices would accept edges from either group, which would block propagation at some point and create a spurious choice point.

On the other hand, restrained sharing reduces spurious symmetries if only for the fact that the co-occurrence of two inflections of one same MWE is less likely. Even if that were satisfactory, it definitely does improve much on storage complexity as compared to no sharing at all, especially in the light of the following new facts.

**Connectives in Optional Constituents and the TNT Effect.** It is worth reminding that Debusmann's solutions are originally targeted at MWEs, and one might argue that their shortcomings are even acceptable taking into account that MWEs occur much less often than self-contained single words. We reply that the magnifying factor of whatever shortcomings a grouping solution may ever have might well be the sheer size of a whole lexicon. In other words, just imagine what if, after all, there were as many group paradigms as there are words or even word senses in a lexicon, or rather, what if virtually every occurrence of any word involved grouping no matter whether it belongs to a MWE. Minor flaws in the grouping scheme might have a disastrous (TNT) effect then.

In order to clearly see how come, it suffices to leave MWEs, assume a generation scenario and consider where connectives (prepositions and conjunctions) in optional constituents (prepositional phrases and subordinate clauses acting as adverbials or noun modifiers) are ever to come from. Even as simple a sentence as "John died for Mary/love" becomes suddenly surrounded with mystery, and it seems rather unlikely that any definitive MWE solution can be formulated before this issue has been suitably tackled. Notice that our focus is not on how the correct connectives are selected (for example, consider the alternations "at ten o'clock/on Monday/in January", which are all time adverbials and only the tip of the iceberg), although that is a very current subject of debate and research. What we are considering is a much more basic issue: the very mechanism allowing their surfacing once they are expected to have no direct semantic counterpart.

Given the current state of the art in XDG, which includes the interesting group technique by Debusmann, we advocate that the hypothesis must be tested that connectives in optional constituents can be generated by some enhanced form of grouping and vertex expansion. Although this surely is in our agenda, we have not yet carried out any such comprehensive test. Instead, we find it essential first to make sure that groups will *scale* – both in terms of instantiability and complexity – or else that simply cannot be the solution.

**Preliminary Evidence.** We proceed to give preliminary evidence that, assuming such a scalable grouping scheme exists, our hypothesis stands a chance. To this end, let us

analyze how the underlined connectives in “Mary knitted it for John while they lived in Paris” could be generated.

Hypothesizing that “for/while” belongs to a group implies asking what its cogrouper(s) must be after all. There seems to be two options only: either “knitted/knitted” or “John/lived”. Generalizing, it is in order to decide whether a connective introducing an optional constituent must group with its governor or its governee, respectively. The second option is the only acceptable, as the resulting groups consist of two components each, namely a connective and (the root of) its governee, while the first option would involve grouping the governor with as many *optional* (i.e. deletable) components as there can be connectives simultaneously governed by the entity at hand (verb, noun, etc.). Not only is that somewhat difficult to determine, but also most of the created components would usually be inactive in most sentences.

Let us hypothetically trace the generation of “for” according to the analysis shown in Fig. 4. Looking up the semantic literal underlying “John” yields two groups, namely $G_1$, a singleton corresponding to a nominative or accusative occurrence of “John”, and $G_2$, corresponding to a prepositioned occurrence. Assuming for clarity that only one-word prepositions are possible, two vertices are created, say *prep* and *john*. Two lexicon entries compete for *john*, one (belonging to $G_1$) accepting either a *subj(ect)* or a *(direct) obj(ect)* edge on the ID dimension and the other ($G_2$) accepting only *pcomp* (preposition complement). As for vertex *prep*, several entries compete for it, one allowing deletion (because $G_1$ does not have two components) plus one $G_2$ entry per possible preposition, accepting, among others, *adv* and donating *pcomp* only to $G_2$ members by means of feature *outgroups*.

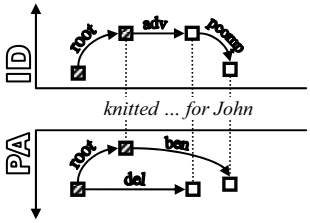



**Fig. 4.** Fragment of an analysis of “Mary knitted it for John”

All that remains to be explained is how “for” (or an equivalent) is to be selected among all other prepositions, or rather, how to state (i) that the specific word “for” is a possible realization of the *ben(eficiary)* relation on dimension PA and (ii) that its ID mother (“knitted”) should be the PA mother of its ID daughter (“John”). For those acquainted with XDG, that immediately suggests some interdimensional principle involving a lexical feature, more precisely some kind of *linking principle*. In fact, the XDK already provides library principle *LinkingDaughterEnd*, which is almost what we need. Given two dimensions $D_1$ and $D_2$ and some function *linkf* of the type given in Eq. (2), it ensures that, for every edge *(src,target,label)* on $D_1$, either there is an edge *(src,target,label')* on $D_2$ such that either $label' \in linkf(src, target, label)$ or $linkf(src, target, label)$ is empty.

For our purposes, a similar principle would suffice operating, though, on groups instead of directly on vertices. Given two dimensions $D_1$ and $D_2$ and some function *link*{, hypothetical principle *LinkingDaughterGroupEnd* would ensure that, for every edge *(src,t,l)* on $D_1$, either (i) there is an edge *(src,t',l')* on $D_2$ such that $l' \in linkf(src,t,l)$ and $groups(t) \cap groups(t') \neq \emptyset$ or (ii) $linkf(src,t,l)$ is empty. Letting $D_1$ = ID, $D_2$ = PA and *link*{ access feature $PAEnd_{\mathrm{ID}}$ of edge targets thus:

$$linkf = \lambda(\_,target,label).\ lex_{ID}(target)('PAEnd')(label), \qquad \textbf{(10)}$$

the application of *LinkingDaughterGroupEnd* ensures the selection of "for", provided that its lexical entry has a value { for $PAEnd_{\mathrm{ID}}$ such that $'ben' \in f('adv')$.

The generation of "while" is perfectly analogous except for the fact that governed finite verbs group with governing conjunctions instead of prepositions. Finally, it is worth mentioning that this grouping solution (governing connectives with governed entities) should only be applied to optional constituents. In contrast, prepositions introducing indirect objects (such as "of" in "approve of") should group with their governing verbs, much like MWEs.

**Consequences.** In the event that the hypothesis introduced above is accepted, grouping will come to play a leading role in XDG praxis. Its status might well be upgraded to that of a primitive. For a start, we have already provided evidence that group-oriented versions of library principles will be needed, and it would be no wonder if the whole original library suddenly became obsolete. In addition, lexicon language will have to be revised to make groups really instantiable, i.e. more friendly to grammar developers. And, as we shall briefly argue in Section 5, so will probably part of its operational semantics, in order to reduce storage complexity.

### 4.3 Instantiability vs. Expansion – Verbs, Nesting, Crossing, and Generalized $N_1$:…:$N_n$ Mapping

Much of what has been discussed for connectives and their generation also applies to auxiliary verbs, even if only to such perfectly grammatical auxiliaries as English "do" in questions and negative sentences. Under our hypothesis for connective generation, it seems reasonable that at least "do" is to be generated by grouping with its main verbs. As all English verbs but a few exceptional cases (auxiliaries and "be") may take this auxiliary, the issues discussed previously are still relevant.

Nevertheless, whether auxiliary or not, verbs are somewhat more complex objects than connectives. A significant portion of their complexity lies in pure syntax and morphology *irrespective* of semantics. For instance, although verbs such as "take", "make", "get", and non-auxiliary "have" may be employed in a variety of senses, most of their syntactic and morphological behaviour remains the same all across. In XDG terms, "have" may group with "do/does/did" and is inflected "has/had/having/to have/etc." no matter whether it should be part of "have a word with" or other MWEs or even work on its own in several alternate senses. Instantiability (i.e. modularity and separation of concerns) here demands that such obvious and productive "irrespectiveness" can be captured, i.e., that partial behaviour can be defined once and for all for later reuse. And as much as possible: it is highly desirable that such very

productive paradigms as grouping with “do” and the “to” infinitive particle could be defined for a whole class of verbs at once. It is worth noticing that the abstract concept of partial and thus reusable specifications already underlies the XDK’s design, although the requirements we have been gathering are not currently met, probably as a result of the underrated status groups have enjoyed thus far.

The described requirements hint at some sort of **group nesting** (e.g. “do have” might be a subgroup inside “[[do have] an argument with]”, as might “on behalf of” in “[[on behalf of] Mary]”), **partial groups** (i.e. underspecified groups, or rather, groups setting a strict subset of all the required features) and the **cross product of complementary partial group paradigms**. The latter refers to the concept that **complementary partial groups** (i.e. setting disjoint subsets of features) might be combined to generate a new group. For example, English phrasal verb “cut off” might have all its morphosyntactic behaviour captured in a partial paradigm *Syn* involving nesting (“[[do cut] off]” and such like). Next, various complementary paradigms $\{Sem_1,\ldots,Sem_n\}$ should specify the possible meanings of “cut off” (“stop”, “separate”, etc.), which might also comprise complex subgraphs. Finally, a hypothetical special cross product $Syn \overline{\times} \bigcup \{Sem_1,\ldots,Sem_n\}$ would conveniently yield all the expected group paradigms, which might, in turn, still be partial and thus reusable for further crossing. Such a scheme, which we have not yet formalized but rather sketch as a teaser, would provide a convenient form of **generalized $N_1$:…:$N_n$ mapping**.

## 5 Future Work – Upgrading Groups and On-Demand Lexicon

The main point of this article has been to resume the discussion on grouping in XDG, give evidence as to how central the issue is to XDG development and gather requirements for enhancing the framework. We believe the way from here is to upgrade groups to the status of a primitive, if not of XDG’s core, at least of the XDK’s lexicon language.

Whether the upgrade makes it to the core or not, one possibility that seems rather promising and unavoidable is modifying the operational semantics of the lexicon component to circumvent storage complexity, among others. In other words, we intend further to exploit the fact that the lexicon component of an implementation does not have directly to reflect its formal counterpart. Specifically, it may well become an *on-demand producer of actual lexicon entries* on an input-by-input basis. This means that all nesting and crossing of primitive group paradigms can be performed on the fly according to the input at hand, which can be relatively easily implemented by means of higher-order programming and is likely to decrease the number of active lexicon entries dramatically even in face of massive grouping.

Such a scheme appears all the more feasible if one takes into account automatic on-the-fly generation and assignment of group ids, every new group occurrence receiving a fresh group id in the scope of the current input, which is rather straightforwardly implemented by means of functional programming and logic variables. This would spare grammar developers from dealing directly with awkward, error-prone group id features and is only possible because groups are very well-behaved: given a group, its edges are either constrained to be internal to itself or free to link to any other group.

Nesting is likely to complicate things a little, but not too much, probably it sufficing to introduce one third option, namely "or constrained to be internal to the innermost enclosing group".

We hope that a lexicon component may thereby reconcile (i) unrestrained sharing (Section 4.2) in grammar development and storage with (ii) neither sharing nor overactivation at all in model generation. All the referred constructs and a vertex expansion algorithm are currently being designed and shall be presented in due time.

## Acknowledgements

This research project has been partially funded by *Conselho Nacional de Desenvolvimento Científico e Tecnológico* – CNPq, a Brazilian government agency fostering technological and scientific development.

## References

1. Brito, L. F. Por uma Gramática de Línguas de Sinais. Tempo Brasileiro Ed., Departamento de Lingüística e Filologia, Universidade Federal do Rio de Janeiro (1995)
2. Debusmann, R.: Multiword Expressions as Dependency Subgraphs. In: Proceedings of the 42nd Annual Meeting of the Association for Computational Linguistics (ACL 2004, "Multiword Expressions: Integrating Processing" Workshop)
3. Debusmann, R., Postolache, O., Traat, M.: A Modular Account of Information Structure in Extensible Dependency Grammar. In: Sixth International Conference on Intelligent Text Processing and Computational Linguistics (CICLING 2005)
4. Debusmann, R., Duchier, D., Kruijff, G. J.: Extensible Dependency Grammar: A New Methodology. In: Proceedings of the 20th International Conference on Computational Linguistics (COLING 2004, Workshop on Recent Advances in Dependency Grammar)
5. Debusmann, R., Duchier, D., Koller, A., Kuhlmann, M., Smolka, G., Thater, S.: A Relational Syntax-Semantics Interface Based on Dependency Grammar. In: Proceedings of the 20th International Conference on Computational Linguistics (COLING 2004)
6. Debusmann, R., Duchier, D., Kuhlmann, M.: Multidimensional Graph Configuration for Natural Language Processing. In: Proceedings of the International Workshop on Constraint Solving and Language Processing (2004) 59–73
7. Duchier, D.: Configuration of labeled trees under lexicalized constraints and principles. In: Journal of Language and Computation (2002)
8. Duchier, D.: Axiomatizing dependency parsing using set constraints. In: Proceedings of the 6th Meeting on the Mathematics of Language (1999)
9. Duchier, D., Debusmann, R.: Topological dependency trees: A constraint-based account of linear precedence. In: Proceedings of the 39th ACL (2001)
10. Duchier, D., Thater, S.: Parsing with Tree Descriptions: a Constraint-Based Approach. In: Sixth International Workshop on Natural Language Understanding and Logic Programming (NLULP 1999) 17–32
11. Gardent, C., Thater, S.: Generating with a Grammar Based on Tree Descriptions: a Constraint-Based Approach. In: Bird, S. (ed.): Proceedings of the 39th Annual Meeting of the Association for Computational Linguistics (ACL 2001)
12. Koller, A., Striegnitz, K.: Generation as Dependency Parsing. In: Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics (ACL 2002) 17-24