On Failure-Driven Constraint-Based Parsing through CHRG*

Veronica Dahl[1], Sinan Eğilmez[2], João Martins[2,3] and J. Emilio Miralles[1]

[1] Simon Fraser University, Burnaby, BC V5A-1S6, Canada,
veronica@cs.sfu.ca, emiralle@sfu.ca
[2] CENTRIA and Departamento de Informática, FCT, Universidade Nova de Lisboa
[3] Computer Science Department, Carnegie Mellon University, Pittsburgh PA
s.egilmez@campus.fct.unl.pt, jmartins@cs.cmu.edu

Abstract. In this article we adapt and further develop a CHRG parsing technique which was initially conceived in the context of grammar induction for Womb Grammars [8]. We also show that applying it to constraint-based linguistic formalisms such as Property Grammars [3] can yield truly direct implementations of specialized parsers that focus on ungrammaticality detection and correction.

1 Introduction

Since the advent of CHR [11] and of its grammatical counterpart CHRG [7], constraint-based linguistic formalisms can materialize through fairly direct methodologies. In this article we study several advantages of doing so. Efficiency-wise, direct CHRG renditions of linguistic constraints promote efficiency by their very adherence to the constraint solving framework. We propose to enhance them further through exploiting our novel methodology, arrived at through our work on grammar induction [8], of checking only for falsification of most constraints in PG [3] and similar formalisms.

Constraint-based theories of grammar are by now widespread in computational linguistics, but there is still no general consensus on what comes under that umbrella.

Shieber's initial characterization of constraint-based theories in terms of common threads such as modularity, declarative constructs and partial information [16] has been influential to this day, partly because it is wide enough to accommodate a variety of grammar theories springing from different fields – linguistics, computational linguistics and artificial intelligence. However such threads are present in many formalisms that would not be viewed as constraint based in the computational science of constraint solving as originally described [17].

The constraint solving paradigm of computing sciences has proved very successful in greatly reducing search spaces by stating a problem in terms of constraints on domain-associated variables, whose possible values are automatically

* This paper was supported by NSERC Discovery grant 31611024, and developed during a visit to Universidade Nova de Lisboa

and successively narrowed down through constraint solving into values that represent a solution, if one exists. It would be therefore most interesting from the point of view of efficiency to consider to what extent the so-called constraint based grammar models fit into the constraint solving model strictly speaking.

The candidate constraint-based theories of grammar to consider, if we follow Shieber's characterization, range widely from augmented transition networks [18] to logic grammars [1], to recent and highly specialized formalisms such as [13], or Womb grammar parsing [8]. Among them, the Property Grammar (PG) framework [3] stands out as an effort to completely describe a grammar in terms of constraints. It defines phrase acceptability in terms of the properties or constraints that must be satisfied by groups of categories (e.g. English noun phrases can be described through a few constraints such as precedence (a determiner must precede a noun), uniqueness (there must be only one determiner), exclusion (an adjective phrase must not coexist with a superlative), and so on). Rather than resulting in either a parse tree or failure, such frameworks characterize a sentence through the list of the constraints a phrase satisfies and the list of constraints it violates, so that even incorrect or incomplete phrases will be parsed to the extent that they can, rather than simply failing.

Because of their sole reliance of constraints, Property Grammars are a main candidate for direct implementation in terms of constraint solving. However the only works which incorporate direct implementation to some extent are [5], [9], [10] and Womb Parsing [8]. Of these, the only one that does not need to calculate all constraints between every pair of constituents is [8]. Instead, it checks constraints only for failure. This works well in the context of grammar induction. In the present paper we examine how to adapt that work to parsing sentences rather than inducing grammars, while retaining both the search space reduction obtained by the failure-driven focus and the direct implementation character (in the constraint-solving sense) of [8].

2 Background

2.1 Property Grammars

The idea of representing a language's grammar solely through properties between constituents was first proposed as a theoretical formalism by Gabriel Bes [2], and reworked by Philippe Blache into Property Grammars [3,4]. Computationally, it relates to Gazdar and Pullum's dissociation of phrase structure rules into the two properties of Immediate Dominance (called constituency in the PG literature) and Linear Precedence (called either precedence or linearity in PG) [12]. It presently comprises the following seven categories (we adopt the handy notation of [10] for readability, and the same example):

Constituency $A : S$, children must have categories in the set S
Obligation $A : \triangle B$, at least one B child
Uniqueness $A : B!$, at most one B child
Precedence $A : B \prec C$, B children precede C children

Requirement $A : B \Rightarrow C$, if B is a child, then also C is a child
Exclusion $A : B \nRightarrow C$, B and C children are mutually exclusive
Dependency $A : B \sim C$, the features of C1 and C2 are the same

This paper will handle full sentences, so that we will generally denote determiners by D, nouns by N, personal nouns by PN, verbs by V, noun phrases by NP, verb phrases by VP and sentences by S.

Example 1. For example, the context free rules $NP \rightarrow D\ N$ and $NP \rightarrow N$, which determine what a noun phrase is, can be translated into the following equivalent constraints: $NP : \{D,\ N\}, NP : D!, NP : \triangle N, NP : N!, NP : D \prec N, D : \{\}, N : \{\}$.

The larger number of constraints allows us to perceive and understand the grammar rules in a more detailed fashion. Furthermore, this finer granularity can be exploited: in some of the literature on PG there is the possibility of declaring some constraints as relaxable. The failure of relaxable contraints is signalled in the output, but does not block the entire sentence's analysis. Implementations not including constraint relaxation capabilities implicitly consider all properties as relaxable.

2.2 Womb Grammar Parsing

Womb Grammar Parsing is a constraint-based parsing methodology developed in 2012 [8], motivated by the need to aid the world's linguist uncover the syntax of the many languages that are not being studied for lack of resources. It was designed to induce a target language's syntax from the known syntax of a source language plus a representative corpus of correct sentences in the target language and the target language's known lexicon. It was presented in two versions: Hybrid Womb Parsing, in which the source language is an existing language for which the syntax is known, and Universal Womb Parsing, in which the source syntax is a hypothetical universal grammar of the authors' own devise, which contains all possible properties between pairs of constituents.

Womb Parsing has the originality of addressing through constraint solving a problem which more usually is cast as a machine learning problem. Additionally, it fully applies the technique of using linguistic information from one language for the task of describing another language, which until then had yielded good results only for specific tasks—such as disambiguating the other language [6], or fixing morphological or syntactic differences by modifying tree-based rules [14]—rather than for syntax induction.

3 From Womb Parsing to Direct PG Parsing

3.1 The Main Idea

Womb Parsing works by adjusting the constraints given in the source grammar until they suit the input corpus. Since this corpus is chosen to be correct and

representative, we're justified e.g. in deleting any constraints that the corpus violates.

For instance, if the source grammar contains the precedence constraint $NP : N \prec ADJ$ and there is an input noun phrase in which an adjective precedes a noun, a CHRG rule will apply and delete the above precedence constraint. Thus, the constraint was checked for violation, not satisfaction.

Each of the properties in PG has similar CHRG rules associated with it. Our proposal here is to adapt these rules into admitting constraint relaxation and signalling failure of non-relaxed constraints that do not satisfy input sentences, rather than adjusting the constraints that failed. Furthermore, the method works on complex sentences rather than just noun phrases.

3.2 Significance

The Womb Parsing method of adapting another language's grammar constraints until they suit the target language's representative input corpus can inspire a new way of viewing the PG *parsing problem*, in which constraints are tested only for failure. In contrast, all previous methods exhaustively test each constraint for all constituents that can participate in it. Concretely, a notion not unlike obligation can be used to identify new phrases, and those phrases can be tentatively expanded from nearby constituents.

Example 2. In "john eats an apple", the noun indicates the existence of a noun phrase. Because noun $NP : \{D, N\}$, then the tentative noun phrase starting at "apple" can include "an" but not "eats", since it can only be constituted by determiners and nouns. This is the constraint-satisfaction criterion for expansion.

For each tentatively expanded phrase, all other constraints are tested for *failure* only. The phrase is allowed to expand only if either no constraint fails, or all constraints that fail have been declared as relaxable. Exhaustive satisfaction check is thus replaced by a smart guided search for a falsifying assignment. This is appropriate provided that the set of satisfied constraints is the exact complement of the set of failed constraints - an assumption that seems reasonable, and that we make. In this case, it follows that we do not need to check the satisfied constraints. Just as for grammar induction we do not need to touch the constraints of the source grammar that do work for the target grammar, we can for PGs only actively check the constraints that do not hold. Should we need to explicitly output those that hold, they could be inferred from the list of constraints that must be satisfied plus those output as unsatisfied, at less computational cost than the usual practice of evaluating all constraints between every pair of constituents, or of adding heuristics to reduce the search space.

This is significant because deep parsing with Property Grammars is theoretically exponential in the number of categories of the grammar and the size of the sentence to parse [15]. Since all previous approaches to PG parsing (except for Womb Parsing) have to calculate all constraints between every pair of constituents, and since the number of failed constraints will in general be much

smaller than the number of satisfied constraints, any parsing methodology that manages to mostly check the failed ones will have a substantial efficiency advantage. If moreover the failed constraints can be checked in the automatic workings of a truly constraint solving formulation like our CHRG one, by means of letting those rules that apply to each particular input sentence trigger, we now have more realistic hopes of turning (our version of) the PG paradigm, which we call Direct PG, into a useful, directly executable constraint-based theory of language. In this first paper we address this hope from the point of view of direct parsing for ungrammaticality detection.

3.3 Phrase Determination

In [8], where we only dealt with noun phrases, constituency did not need to be checked explicitly, because it followed from a lexicon made only from noun phrase constituents. The core idea was to start the noun phrase from its head noun, and try to expand it. Expansion would fail if any of the grammatical constraints failed.

More concretely, one expands instantiated categories, which are CHRG predicates of the type `iCat(Start, End, CategoryType, Attributes, Tree)`. Instantiated categories simply keep track of the location, attributes (gender and plurality) and parse tree of all categories, or phrase types, in a specific sentence (e.g. N, NP, etc).

Example 3 (Instantiated categories). Take the noun phrase "an apple". Parsing it results in the following instantiated categories.

```
iCat(0, 1, det, [sing,neutral], det(an))
iCat(1, 2, n,   [sing,neutral], n(apple))
iCat(0, 2, np,  [sing,neutral],
  np(
    iCat(0, 1, det, [sing,neutral], det(an)),
    iCat(1, 2, n,   [sing,neutral], n(apple))
  )
)
```

Notice that the NP inherits the attributes of the underlying N, thus implementing their dependency constraints; and that a tree is built as a side-effect from parsing.

Since in this work we need to treat whole sentences rather than just noun phrases, some additional considerations are necessary:

- Whereas in [8] it was implicit that noun phrases grew from their head nouns, for full sentences each phrasal category must explicitly have its own *head*. These heads follow closely from the grammar itself and the obligation constraints it implies. In our case, we have:

  ```
  head(n, np). head(pn, np). head(v, vp). head(vp, sentence).
  ```

– For full sentences, the extra complexity requires phrases to be treated in a specific ordering. For example, because a verb phrase can contain noun phrases, noun phrases should be parsed before verb phrases, and verb phrases should be parsed before sentences. The ordering is currently specified as part of the grammar, though it could be induced from the directed constituency graph implied by a grammar. For the sentences we consider, the following order is used:

```
parseOrder([np,vp,sentence]).
```

After all categories of the type at the head of the list have been parsed and expanded, the algorithm moves on to the next category.

```
parseOrder([_|L]) <=> parseOrder(L).
```

– While the PG formalism per se does not care about syntactic trees (it just characterizes an input sentence through its lists of satisfied and unsatisfied properties), they are important for complex sentences, and are thus built explicitly by the algorithm. Not only is this handy to the traditional way in which linguists are used to thinking of a parse result, but it is also useful to ensure that the constraints apply on immediate daughters only. For instance, consider the parse tree of "john eats an apple" in Figure 1. It contains two noun phrases, "john" and "an apple", but only 'john" is a direct daughter of the sentence ("an apple" is a direct daughter of the verb phrase "eats an apple"). Therefore, the uniqueness of a noun phrase NP in a sentence S, $S : NP!$ does not fail, since only the direct daughter noun phrase "john" is identified as the sentence's noun phrase.

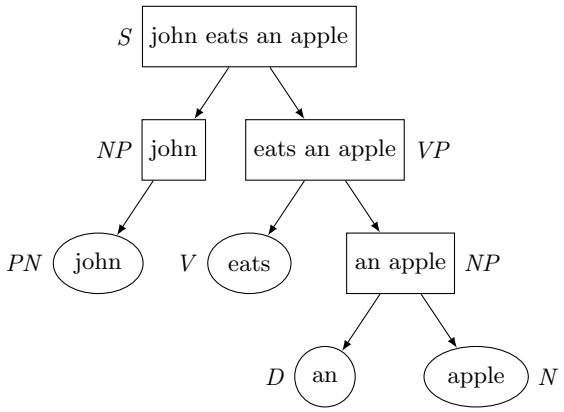

Fig. 1. Parse tree of the sentence.

Taking all these considerations into account, the rule for creating new categories from their heads is the following.

```
iCat(N1, N2, Comp,Attr,Tree), {parseOrder([Cat|_])}
  ::> head(Comp, Cat), NewTree=..[Cat,iCat(N1,N2,Comp,Attr,Tree)]
  | iCat(Cat, Attr, NewTree).
```

This rule is found after the expansion and constraint satisfaction checks. This means it will only fire when `iCat(N1,N2,Comp,Attr,Tree)` is maximally expanded, it does not violate unrelaxable constraints. After the rule fires, there will be a new instantiated category, which will in turn be expanded maximally, and checked against the constraints. Once no more checks or expansions can be done on any category of the current type, the parsing moves on to the next category type given by `parseOrder`\`1`.

The expansion rules are a little more verbose, but the reader should now be acquainted with most predicates used. We present here a simplified version of the left-expansion rule (the right-expansion rule is symmetric).

```
  !iCat(N1, N2, Comp, Attr1, Tree1),
  iCat(N2, N3, Cat, Attr2, Tree2),
  !{tpl(constituency(Cat, L))},!{evil(EL)},
  !{parseOrder([Cat|_])}
 <:> member(Comp, L),
     buildTree(Cat, iCat(N1,N2,Comp, Attr1, Tree1),
                    iCat(N2,N3,Cat , Attr2, Tree2),
                    Tree)
 | iCat(N1, N3, Cat, Attr2, Tree).
```

Expansion considers two adjacent categories only. The component category will not be removed (e.g. the D of an NP), whereas the `iCat` of the expanding category is replaced by its expanded version. This rule is guided by constituency, so that senseless expansions do not occur, e.g. NP can be expanded to include an adjacent D (to the left or right!) but not a V. A consequence of this approach is that the constituency constraint is not relaxable.

Notice that the only constraint checked here is constituency. The other constraints are deferred until after the expansion. Whenever an expansion causes some constraint to be falsified, a rule will fire to alert us to that effect. If the falsified constraint is relaxable, the algorithm is allowed to continue. If it is not, however, the expansion that caused the falsification is retracted, and reverts back to its unexpanded predecessor.

We now present a couple of simplified examples of such constraint violation rules for the reader's benefit. We first show the rule that checks for the violation of uniqueness, $Cat : C$!:

```
  iCat(N1, N2, C, Attr1, Tree1), % Found a C
  ...,                           % and sometime later...
  iCat(N3, N4, C, Attr2, Tree2), % Another C
  {iCat(N5, N6, Cat, _, Tree)},  % Found a Cat
  {tpl(unicity(Cat, C))}         % Unicity constraint
  ::>
```

```
% The C's are within the bounds of Cat
      N5 =< N1, N4 =< N6,
% And they are its direct daughters
      Tree=..[Cat|T],
      member(iCat(N1, N2, C, Attr1, Tree1), T),
      member(iCat(N3, N4, C, Attr2, Tree2), T)
  | falsify(uniqueness(Cat,C)):(N1,N4). % Uniqueness falsified!
```

This rule can fire even if a category hasn't been fully expanded - adding more components to an `iCat` is not going to modify uniqueness having been violated. We now present obligation, $\mathit{Cat} : \triangle C$, whose rule is structured differently.

```
   iCat(N1, N2, Cat, Attr, Tree),     % Found Cat
   {tpl(obligation(Cat, C))}          % Cat should have C
  ::> Tree=..[Cat|T],                 % Get children
      not(member(iCat(_,_,C,_,_), T)) % There isn't a child C
  | falsify(obligation(Cat, C)).      % Obligation is violated!
```

The obligation rule checks a given category for its direct daughters. Since `iCat`s are retracted when they fail, this check only fires after the `iCat` has been fully expanded. Notice the difference with the uniqueness rule discussed above. There is, in fact, an interesting insight regarding the different natures of these rules, which will be expounded upon in Section 3.5.

3.4 Algorithm overview

Since the specifics of the algorithm can get a little involved, it is perhaps helpful to abstract away the rules, their positions and the order in which they fire. In fact, the algorithm is conceptually extremely simple.

1. If there is an instantiated category of the type being handled (e.g. NP, VP, S) that can be expanded without violating unrelaxable constraints, the expansion is carried out. Violated relaxable constraints are stored for reference.
2. If no more expansions are possible and there is an instantiated category can be a head for the category being handled, then a new instantiated category of that type is created from the head.
3. If no more expansions are possible and there are no more potential heads, the algorithm proceeds to handle the next category type.

Example 4. Consider our running example of "john eats an apple" again and the following context free grammar (stated in bottom-up fashion), $PN \rightarrow NP$; $D, N \rightarrow NP$; $V, NP \rightarrow VP$; $NP, VP \rightarrow S$, where PN is a proper noun and S a sentence. N and PN are the heads of NP, V of VP and VP of S.

The parse ordering associated with these rules is NP, VP, S. We will consider the constraints induced from the grammar rather than the grammar itself.

The algorithm thus starts by creating the leaf instantiated categories for the PN, N, D, and V words. Then, the NPs "john" and "apple" are created from the respective PN and N heads. The NP "apple" is expandable to "an apple", and then no more expansions are possible for "john" or "an appple', since $NP : \{D,\ N\}$.

The algorithm thus moves to VPs, and create one from the V head "eats". Since $VP : \{V,\ NP\}$, the VP is expandable to both left, for "john" and right, for "an apple". The right expansion succeeds without problems. The left expansion falsifies the precedence $VP : V \prec NP$, and is therefore not carried out.

Since no more expansions are possible for VPs, a S is created from the VP and expanded to "john", creating the full sentence. However, suppose that $VP : V \prec NP$ were relaxable. Then, the VP would expand both left and right, encompassing the whole sentence. With no more expansions possible, a sentence S would be created from the VP containing the whole sentence, thereby violating the obligation $S : \triangle NP$ (since the constraint requires NP as a direct *direct* daughter), and would thus be retracted. This parse would not result in a sentence. If that restriction were, in turn, relaxable, the parse would keep the S instead of retracting it.

If no constraints were relaxable, "john eats apple" would not parse a sentence S (or a VP or even an NP) because nouns must have determiners, i.e. $NP : N \Rightarrow D$. If that were relaxed, however, the entire sentence would be parsed with no further complications! This highlights the flexibility of the approach.

3.5 Types of constraints, and their procedural implications

As we have seen, our parsing algorithm's reliance on expansion from phrasal heads to incorporate allowable constituents for that phrase places the constraints of constituency and head obligation in the category of non-relaxable, with all other constraints being relaxable.

For relaxable constraints we can identify a further, useful distinction: that between permanent and changeable constraints. Notice that CHRG rules for constraints such as precedence can be allowed to apply immediately after the two constituents they involve show up in the constraint store, no matter what stage of parsing we are at. This is because as soon as the constraint can be evaluated, its value is *permanent*. Once it fails, adding new constituents to the category cannot make it succeed, since e.g. two unordered elements will remain unordered even when adding more elements around them. Accordingly, our methodology only checks for failure of permanent constraints, regardless of at what stage they fail.

Other constraints, which we call *changing* constraints, can change in their evaluation by the incorporation of one more category into a given phrase for which, without this added category, the constraint had an opposite value. Thus, for "a red apple", at the point in which "red" and "apple" have been grouped into a noun phrase, this noun phrase will fail the constraint that a noun requires a determiner, $NP : N \Rightarrow D$. However subsequent expansion of the noun phrase into "a red apple" will remedy the failure. This subtype of changing constraints is called *recoverable*. For recoverable constraints, we simply wait until a phrase

has been maximally expanded before testing whether they fail. We effect this by applying all expansion rules before checking any recoverable constraints.

For the other type of changing constraints, called *filtering* constraints, the addition of one more element into a phrase might make the constraint fail where it succeeded before. As an example, while unicity of determiner holds for "the book", it no longer holds for "the the book". Since the rules that check this type of constraint are triggered only when the constraint fails (e.g. when two determiners show up inside a noun phrase), there is no need for waiting mechanisms, or for any maximal expansion to precede their checking. In this sense our approach is much more focused than previous ones, as a consequence of better fitting the constraint-solving model.

3.6 Constraint Relaxation

Constraint relaxation is indicated by the user as part of the grammar's definition, through a special system predicate `relaxable(L)`. Any constraints inside the list `L` can be relaxed; the others cannot. If all the *permanent* constraints that are falsified during a phrase's expansion are relaxable, the constraints are simply added to a list of relaxed constraints and the expansion is accepted. *Changing* constraints are handled slightly differently. As seen, these should only be evaluated once the phrase cannot be expanded anymore. If, at that stage, all falsified changing constraints are relaxable, then the phrase is accepted, and furthermore allowed to be considered as a head of a more complex phrase.

4 Concluding Remarks

We have presented a novel parsing methodology inspired from Womb Grammar Parsing which greatly reduced the combinatorial explosion inherent in PG parsing by three main contributions: a) a truly direct constraint-solving materialization of constraints, b) a great reduction of the number of constraints to be evaluated, through expressing most constraints in terms of failure only, and c) relaxation handling, which is not always present in previous PG renditions.

While a formal comparative complexity analysis would require a previous underlying analysis of the disparate implementation means of the various formulations, we can intuitively trust that our two main contributions above mentioned are bound to enable a much greater degree of efficiency than was previously possible. Roughly speaking, [9] encodes the input PG into a set of CHRG rules that directly interpret the grammar in terms of satisfied or relaxed constraints, which are then propagated while a syntactic tree is built as a side effect. For efficiency, the implementation controls the way in which a constraint is selected for evaluation, but still, the failure or success value of all constraints among every pair of constituents is exhaustively calculated or propagated. [5] resorts to heuristics but remains highly combinatorially explosive, since it involves developing a complete table of satisfied and unsatisfied constraints for every pair of categories involved. [10] is even more combinatorially explosive but interestingly, develops

a parsing architecture with full reliance on constraint satisfaction, using classical constraint-based techniques such as branch-and-bound to select and propagate constraint evaluations. It also allows for the relaxation of constraints, by choosing those solutions that maximize the ratio between satisfied and unsatisfied properties - there is no control over which properties are relaxable, however.

The approach in [9], like our own, also completes original assignments of categories with new categories when they are inferred, and it exploits a similar distinction between permanent and changing constraints to minimize recalculation. However, they need to explicitly inherit permanent constraints at each step, and to recalculate and eventually update at each step the satisfaction value of changing constraints. In contrast, as we have seen, our parsing methodology allows us to simply bypass the need to inherit the value of permanent constraints (through only checking them on maximal phrases), to postpone the checking of recoverable constraints until the phrase has been identified as maximal, and to only check for filtering constraints when and if they do fail. Also in this sense we obtain considerable efficiency gains.

References

1. Abramson, H., Dahl, V.: Logic grammars. Symbolic computation: artificial intelligence, Springer (1989)
2. Bes, G.G., Hagege, C.: Properties in 5p. Tech. rep., Universite de Clermont Ferrand (2001)
3. Blache, P.: Property grammars: A fully constraint-based theory. In: Christiansen, H., Skadhauge, P.R., Villadsen, J. (eds.) CSLP. Lecture Notes in Computer Science, vol. 3438, pp. 1–16. Springer (2004)
4. Blache, P., Balfourier, J.M.: Property grammars: a flexible constraint-based approach to parsing. In: IWPT (2001)
5. Blache, P., Morawietz, F.: Some aspects of natural language processing and constraint programming. Tech. rep., Universitat Stuttgart, Universitat Tubingen, IBM Deutschland (2000)
6. Burkett, D., Klein, D.: Two languages are better than one (for syntactic parsing). In: EMNLP. pp. 877–886. ACL (2008)
7. Christiansen, H.: CHR grammars. TPLP 5(4-5), 467–501 (2005)
8. Dahl, V., Miralles, J.: Womb grammars: Constraint solving for grammar induction. In: Sneyers, J., Frühwirth, T. (eds.) Proceedings of the 9th Workshop on Constraint Handling Rules. vol. Technical Report CW 624, pp. 32–40. Department of Computer Science, K.U. Leuven (2012)
9. Dahl, V., Blache, P.: Directly executable constraint based grammars. Proc. Journees Francophones de Programmation en Logique avec Contraintes (2004)
10. Duchier, D., Dao, T.B.H., Parmentier, Y.: Model-Theory and Implementation of Property Grammars with Features. Journal of Logic and Computation p. 19 (2013), `http://hal.archives-ouvertes.fr/hal-00782398/en/`
11. Frühwirth, T.W.: Theory and practice of constraint handling rules. J. Log. Program. 37(1-3), 95–138 (1998)
12. Gazdar, G.: Phrase structure grammars and natural languages. In: IJCAI. pp. 556–565 (1983)

13. Muresan, S.: A learnable constraint-based grammar formalism. In: Huang, C.R., Jurafsky, D. (eds.) COLING (Posters). pp. 885–893. Chinese Information Processing Society of China (2010)
14. Nicolas, L., Molinero, M.A., Sagot, B., Trigo, E.S., de La Clergerie, E., , Farré, J., Vergés, J.M.: Towards efficient production of linguistic resources: the Victoria Project (2009)
15. van Rullen, T.: Vers une analyse syntaxique à granularité variable. Ph.D. thesis, Université de Provence (2005)
16. Shieber, S.M.: Constraint-based grammar formalisms - parsing and type inference for natural and computer languages. MIT Press (1992)
17. Waltz, D.: Understanding line drawings of scenes with shadows. In: The Psychology of Computer Vision. p. pages. McGraw-Hill (1975)
18. Woods, W.A.: Transition network grammars for natural language analysis. Commun. ACM 13(10), 591–606 (Oct 1970), http://doi.acm.org/10.1145/355598.362773

A Appendix: Code, examples

For reviewing purposes, the code is available at the following repository: `http://www.cs.cmu.edu/~jmartins/ng.pl`. It was implemented using SWI-Prolog (`http://www.swi-prolog.org/`) and CHRG (`http://akira.ruc.dk/~henning/chrg/`). The suggested way of running it is by installing SWI-Prolog, downloading our source code and the CHRG source and putting them in the same folder. Then, run SWI-Prolog using `swipl`, and at the prompt `?- compile(ng).`. To parse phrases, use, for instance, `?- doParse([jean, mange, une, pomme]).` The output is hopefully self-evident, but the most important predicates are `iCat(left_bound, right_bound, phrase_type, attributes, syntax_tree)`, which represents phrases, and of which you should feel free to ignore the last, very verbose argument, containing the syntax tree. Then, `evil/1` contains the list of retracted `iCat`s (due to constraint falsification), and the `unsat/1` contains the list of falsified but relaxed constraints.

A.1 Grammar and suggested examples

The grammar can be found in the `ng.pl` file, under the predicate `init_grammar`, which initialises it. It contains a few suggested relaxable predicates, which should be run with the above example, `?- doParse([jean, mange, une, pomme]).`. Another suggestion is running `?- doParse([jean, mange, pomme, une]).` with no relaxable predicates, and running it with the suggested relaxation of `precedence(np,det,n)`, also in the file. This will show that a failure at the very basic *NP* level will not interfere with further parsing. Other suggestions are `?- doParse([jean, mange, pomme]).`, by relaxing the requirement constraint between nouns and determiners, or `?- doParse([le, jean, mange, une, pomme]).` and the related exclusion constraint between *PN* and *D*. It is also interesting to add or remove the constraint that *VP* should have *NP* (commented in the file), causing parsing to stop with `pomme, une` or just `pomme` with no determiner.