A New Linguistic Engine for NooJ: Parsing Context-Sensitive Grammars with Finite-State Machines

Max Silberztein(✉)

Université de Franche-Comté, Besançon, France
max.silberztein@univ-fcomte.fr

Abstract. NooJ is a linguistic development environment that allows linguists to construct large linguistic resources of the four types in the Chomsky hierarchy. NooJ uses a bottom-up, "cascade" approach to sequentially apply these linguistic resources: each parsing operation accesses a Text Annotation Structure, and enriches it by adding or removing linguistic annotations to it. We discuss the drawbacks of this approach, and we present a new approach that requires that all NooJ linguistic resources be represented by a single type of finite-state machine. In order to do that, we must solve theoretical problems such as "how to handle Context-Sensitive Grammars with finite-state machines", as well as some engineering problems such as "how to compose sets of large dictionaries and grammars into a single finite-state machine". Our first experiments show that although that composing large finite-state machines is extremely costly theoretically, the fact that linguistic resources in a typical NooJ cascade depend on each other heavily keeps the size of all intermediary machines manageable. Once the final resulting finite-state machine has been compiled and loaded in memory (e.g. on a webserver) it can be used to parse large texts in linear time.

Keywords: NooJ · RA · Recursive automata · Linguistic engine

1 Introduction

Formal Grammars have been introduced to linguists by (Chomsky 1956) as mathematical tools to describe languages. A Formal Grammar is a set of production rules such as $\alpha \rightarrow \beta$ where both α and β are sequences of empty strings (noted ε), terminal and non-terminal symbols. Depending on constraints on the nature of α and β, (Chomsky 1956) classifies Formal Grammars in four increasingly powerful types: Regular, Context-Free, Context-Sensitive and Unrestricted Grammars. As the power of grammars is augmented, the efficiency of the corresponding parsers is degraded:

S. Mbarki et al. (Eds.): NooJ 2017, CCIS 811, pp. 240–250, 2018.
https://doi.org/10.1007/978-3-319-73420-0_20

parsers for RGs can run in linear time,[1] parsers for CFGs (e.g. CYK) can run in cubic time,[2] parsers for CSGs run in exponential time,[3] and the halting problem of parsers for UGs is undecidable.[4]

Linguists have designed formal notations — i.e. formalisms — to help construct these four types of grammars. For instance, XFST and its variants HFST and SFST[5] are well adapted to the design of RGs; it is straightforward to write CFGs with GPSG;[6] LFG[7] allows linguists to construct CSGs, and HPSG can handle UGs.

Ideally, a linguist would pick each type of formalism according to each type of linguistic phenomenon: for instance, one could use RGs to describe spelling variants, CFGs to compute the structure of sentences and CSGs to describe agreement and distributional constraints. Unfortunately, the aforementioned formalisms are not compatible, making it impossible to include an XFST grammar and a GPSG grammar inside a LFG grammar, or even merge a TAG grammar with a CCG grammar.

NooJ[8] was developed for this reason: it provides linguists with a unified formalism to handle the four types of grammars, thus ensuring that all linguistic resources are compatible. With NooJ, a CFG is a RG that contains recursive calls; a CSG is a CFG that contains variables and contextual constraints; an UG is a CSG that uses its outputs to perform transformation operations on its input.

Typically, a NooJ analysis consists in sequentially applying a set of linguistic resources to a text in a bottom-up approach: dictionaries (represented by acyclic Finite-State Automata in which terminal states point to the recognized word's analysis), morphological and local syntactic grammars (Recursive Finite-State Transducers), structural syntactic and/or semantic grammars (Enhanced Recursive Transducers to handle CSGs and UGs). The most ambitious NooJ applications combine grammars from of all four types to perform transformational analyses (e.g. to compute "It is not Lea who is loved by Joe" from "Joe loves Lea [Passive] [Cleft1] [Negation]"),[9] or to perform automatic text translation from one language to another.[10]

Although NooJ's approach has been proven to be useful for a number of applications (mostly, corpus linguistics where the size of a 'large' corpus is a hundred

---

[1] The time it takes to parse a text is proportional to its length *n*, i.e. *O*(n).

[2] See for instance (Kasami 1965).

[3] See for instance XLFG which is a parser for the LFG formalism.

[4] i.e. one cannot even predict if a Turing machine will parse any text in finite time. Most linguists doubt that we would need the power of a Turing machine to describe real world natural languages. (Silberztein 2016a) argued that the typical examples of phenomena that would require unrestricted grammars are "extra-linguistic" in nature (e.g. anaphora resolution).

[5] See (Linden et al. 2010), (Schmid 2005) and (Karttunen et al. 1997).

[6] See (Gazdar 1988).

[7] See (Kaplan Bresnan 1982) and (Dalrymple 1995).

[8] See (Silberztein 2016a). NooJ is a free, open-source linguistic development environment available at www.nooj-association.org and supported and distributed by the European Metashare platform.

[9] (Silberztein 2016b) shows how NooJ produces several millions of transformational variants for the simple sentence "Joe loves Lea".

[10] Translations are performed just like transformations; the only difference being that the translated lexemes are obtained via a lookup of a multilingual dictionary.

megabytes at most), it does not scale up well, and NooJ cannot process large amount of texts to search the WEB for instance, or filter out all tweets in real time (100,000 per minute).

From a computational point of view, the different types of machines (.nod files for dictionaries, .nof for inflection, .nom for morphology, .nog for syntax) have imposed different parsers and a complex architecture, thus making it impossible to combine machines and thus to perform some global optimization.

2 The RA Approach

The basic principle behind the new RA linguistic engine is to unify all machines, so that only one parser will be used to perform all analyses. Thereafter, it will be possible to merge and compose them instead of applying them in sequence. For that to happen, we need to solve a series of problems, among them the following:

- how to represent morphological rules so that they can be used both to generate forms and to lemmatize them;
- how to represent CFGs and CSGs with finite-state machines.

2.1 Reversible Morphological Grammars

In NooJ, all linguistic resources are supposed to be application-neutral; in particular, they should be useable both by parsers and by generators. Although this is indeed the case at the functional level, internally NooJ's morphological parser and generator work quite differently: NooJ inflects and derives words by applying morphological recursive transducers to lemmas, but lemmatizes forms by looking up automata of a dictionary. Morphological paradigms are described by enhanced CFGs such as the following:

TABLE = <E>/singular | s/plural;

Paradigm TABLE states that if one adds the empty string (<E>) to a lemma (e.g. *pen*), the resulting word form (e.g. *pen*) is in the singular; if one adds an 's' to the lemma, the resulting word form (e.g. *pens*) is in the plural, i.e.:

pen + <E> → pen/singular
pen + s → pens/plural

From the following dictionary entry:

pen, NOUN + FLX = TABLE

An exploration of the transducer compiled from rule TABLE produces the following output:[11]

pen, pen, NOUN + FLX = TABLE + singular
pens, pen, NOUN + FLX = TABLE + plural

[11] The input/output result produced by the corresponding RA Finite-State Machine is underlined.

All the forms generated by morphological transducers are then stored in an acyclic finite-state automaton built using a variant of (Daciuk et al. 2000)'s linear minimization algorithm. However, the fact that the resulting automaton needs to store the original lemma (e.g. *pen*) for each of its entries degrades the efficiency of the minimization algorithm considerably.

Rather than using a transducer (the morphological grammar) to produce inflected forms, and an automaton (the dictionary) to lemmatize forms, RA uses only one machine in both directions. To lemmatize an inflected form, RA applies the machine compiled from grammar TABLE "in reverse":

Pen – <E> → pen
Pens – s → pen

Since we can now compute the lemma from each of its inflected forms, it is no longer necessary to store the lemma in the dictionary: therefore all the forms that share the same analysis (e.g. *beds*, *cars*, *engines*, etc.) will be associated with one unique terminal state. However, it is not always possible to simply "reverse" a morphological grammar. For instance, consider the following paradigm:

MAN = <E>/singular | <B><B>en/plural;

The <B> operator (for "Backspace") deletes the letter located on top of the word stack (i.e. at its end). This paradigm produces the two forms *man* and *men* from the lexical entry *man*:

man + <E> → man/singular
man + <B><B>en → men/plural

If we simply reverse it, we get:

man – <E> → man
men – ne<B><B> → m??

The generation process used the <B> operator to delete a letter, but the lemmatization process doesn't know which letter was deleted in the first place. More generally, NooJ morphological grammars are not reversible.

The new RA framework thus proposes to replace the former NooJ grammar with the following one:

MAN = <E>/s | <Bn><Ba>en;

i.e. one adds to the former destructive NooJ operators the information required to reverse them. The new grammar can then be used both to generate inflected forms and to lemmatize them:

man – <E> → man
men – ne<Ba><Bn> → man

Just like all other linguistic resources, RA's morphological grammars can be used both to generate all the inflected forms from a given lemma, and to lemmatize every inflected form.

Finally, we note that during the process of inflecting a lemma using initial NooJ's operator <B>, one knows exactly what letter is being deleted (it is on top of the stack): an automatic compiler can then use this information to compute the RA grammar equivalent to the initial NooJ grammar, just by simulating the inflection process.

2.2 Context-Free Grammars

In Context-Free Grammars, $\alpha \rightarrow \beta$ rules are such that α contains one and only one non-terminal symbol, whereas β can contain any sequence of terminal and non-terminal symbols. In NooJ, CFGs are written either in a text form:[12]

$$\alpha = \beta;$$

or graphically, by a set of recursive graphs. For instance, Fig. 1 presents a grammar that contains 3 graphs: the top graph (called Main) contains references to graph NP and graph VG.

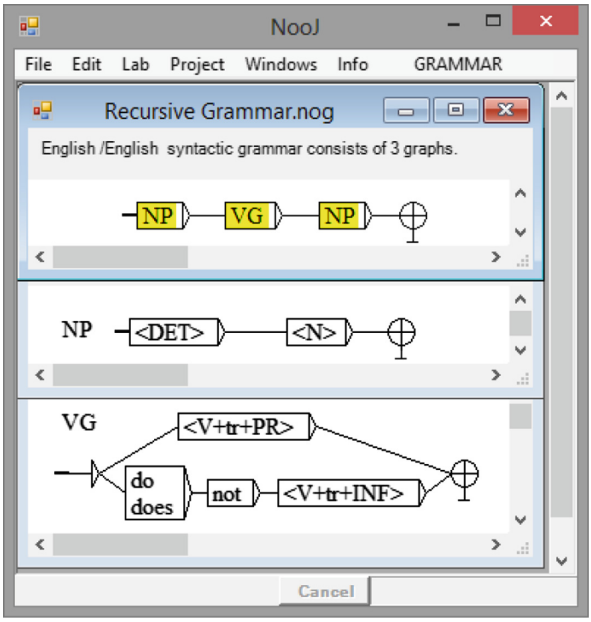

Fig. 1. A graphical CFG

The RA linguistic engine 'flattens' recursive graphs by removing left and right recursions. Middle recursions typically correspond to embedded structures such as in the following sentences:

[12] In NooJ CFG grammars, β is a regular expression, built on terminal and non-terminal symbols and <E> (empty string), e.g.: NP = (<DET> | <E>) <ADJ> * <NOUN>.

The cat saw the mouse
The cat (Joe likes) saw the mouse
?*The cat (the cousin (Eva talked to) likes) saw the mouse
*The cat (the cousin (the neighbor (you know) talked to) likes) saw the mouse

Although there are over 30 languages that are currently being formalized with NooJ by their native speakers, we are yet to find good examples of unlimited middle recursions that feel natural beyond two levels of embedded phrases.

The new engine RA 'cheats' by replacing all middle recursion references in a graph with a fixed number (typically two) of copies of the corresponding embedded graph. As a consequence, the last sentence of the previous example would not be recognized by RA's parser. However, the benefit is that RA can compile any NooJ CFG grammar into a finite-state machine. If the size of the initial NooJ grammar is n, the size of the corresponding RA grammar could be n^3 in the worst case; in practice though, all the NooJ grammars we encountered produced RA grammars that were lower than eight times the size of the initial NooJ grammars.

2.3 Context-Sensitive Grammars

In Context-Sensitive Grammars, production rules are of the type $\gamma A \rightarrow \gamma\delta$, where A is a non-terminal symbol, and δ and γ are sequences of non-terminal and terminal symbols (γ describes the context in which A can be rewritten into δ).[13] These rules can be rewritten in NooJ by adding the following rules:

$R_i \rightarrow R_i$ ($context γ) A
A $\rightarrow$ δ/<$context>

Where rules R_i are added for each rule in the original grammar; these rules are used to define a $context variable that will be set every time context γ appears before A. The second rule uses constraint <$context> to check that variable $context has indeed been defined (if not, the rule is rejected and A is not rewritten as δ).

Although this translation shows that variables and constraints give NooJ the power to process any CSG, using variables and constraints is much more natural in practice. (Seljan et al. 2002) gives several examples of typical formal CSGs, e.g. $\{a^n b^n c^n d^n, n > 0\}$, $\{a^n b^m c^n d^m, n > 0\}$, etc. (Silberztein 2016a) shows how to translate them into NooJ graphs that are much easier to understand. For instance, the following grammar recognizes languages such as $\{a^n b^n c^n d^n e^n, n > 0\}$:

Without taking into account the constraints shown in bold in the graph, this grammar recognizes the regular language a*b*c*d*e*, storing the sequence of a's in variable $A, the sequence of b's in variable $B, the sequence

[13] This is the definition of left context-sensitive grammars. In right context-sensitive grammars, the non-terminal symbol of the left hand side is followed by the context, i.e. production rules look like: $A\gamma \rightarrow \delta\gamma$. The equivalence of left and right context-sensitive grammars was established by (Penttonen 1974). Another, more general definition is that context-sensitive grammars contain rules such as $\gamma A\gamma' \rightarrow \gamma\delta\gamma'$. (Kuroda 1964) proves that all these grammars have the same power of description.

of d's in $D and the sequence of e's in $E. Thereafter, the parser checks that properties $LENGTH for the four variables are identical. Note that "mildly context-sensitive formalisms" such as CCG or TAG formalisms can represent languages such as $\{a^n b^n c^n, n > 0\}$ but not the one in Fig. 2.

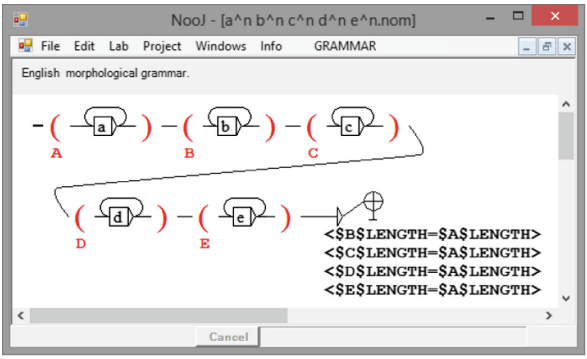

Fig. 2. $\{a^n b^n c^n d^n e^n, n > 0\}$

Figure 3 is an example close to linguists' needs:

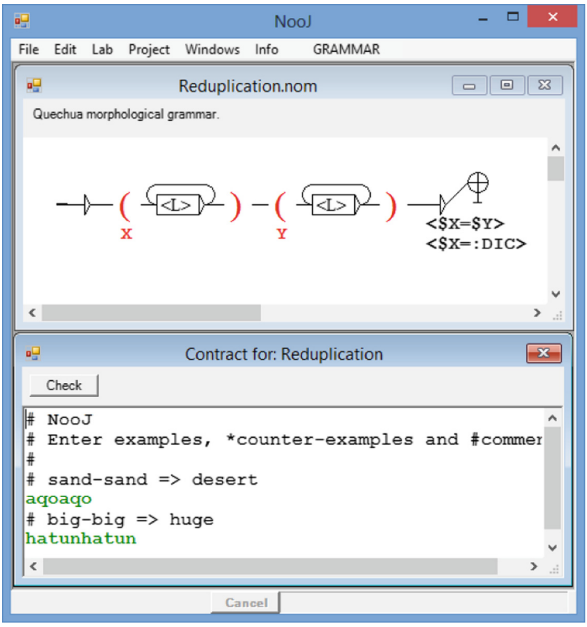

Fig. 3. Reduplication

<L> matches any letter. This morphological grammar recognizes word forms that contain the same affix twice (<$X=$Y>), and also that the affix is a valid lexical entry (<$X=:DIC>). For instance, in Quetchua, the word "ago" [sand] can be derived in "agoago" [sand-sand = desert]. Many languages use reduplications, such as Indonesian, Tagalog, Japanese, Mandarine, Quechua, etc.

As for CFGs, RA can compile a NooJ CSG into a corresponding deterministic finite-state machine. RA parses a text of length *t* by applying the finite-state machine in two steps:

- During the first step, RA ignores all constraints; this parsing can be performed in *O*(t).
- The parsing process may produce multiple intermediary solutions;[14] the number of solutions is *O*(t).
- The RA engine needs to validate each solution by checking the corresponding constraints.

NooJ constraints are of three types:

- existence (e.g. <$context>) to check if a variable has been set or is undefined;
- string equality (e.g. <$X=$Y>) to check if two lexemes are equal;
- symbol matches (e.g. <$X=:VERB>) to check if a lexeme matches an annotation and its properties.

All constraints are implemented by simple unification operations like the ones in systems such as LFG or HPSG, but they are not recursive (because Nooj's annotations are 'flat' sequences of atomic property/value pairs, rather than trees). Therefore, checking each constraint can be performed in constant time; the maximum number of constraints to check for a solution is proportional to the size of the grammar *g*; therefore, RA can apply CSGs to texts in *O*(g t).

2.4 Compiling Syntactic Grammars

In NooJ, there is a fundamental difference between grammars that operate inside word forms at the character level, and grammars that operate at the phrase/sentence level at the lexeme level. In consequence, NooJ uses two different parsers and two different machines (.nom and .nog).

RA can process NooJ morphological and orthographical grammars with a simple format conversion; however, transforming a NooJ syntactic grammar into a RA grammar necessitates human intervention. Consider the graph of Fig. 4 below.

- All the connections to or from an empty node should be translated into ε-transitions;
- All the connections from an English word to another one (e.g. from "April" to "the") must be translated into transitions labeled with a space character " ";

[14] This is the case for the grammar of Fig. 3, for which the parsing process produces *w-1* intermediary solutions because there are *w-1* ways to split a word form into two non-empty affixes.

– Connections from an English word to some punctuation mark (e.g. “Monday” to “,”) must be translated into ε-transitions; connections to other punctuation marks (e.g. “—”) require spaces.

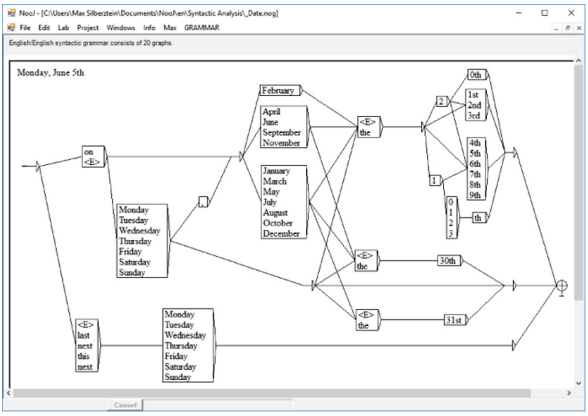

Fig. 4. A syntactic grammar

Choosing between spaces and ε-transitions is not straightforward around symbols because a given NooJ symbol (e.g. <VERB>) can refer both to a single lexeme (e.g. “eating”) or to a sequence of word forms (e.g. “is willing to stop eating”). Lastly, NooJ recursive grammars might include embedded graphs that recognize the empty string: to translate a NooJ syntactic grammar into a pure RA grammar (at the character level) involves computing all the *Follow Sets* of a grammar, i.e. the sets of prefixes that are recognized after each embedded graph.

Note finally that these translation rules depend on the text language; for instance, the colon character must follow a space in French, but not in English; contracted words such as *cannot* for <can> <not>) are exceptions that must be taken into account; agglutinative languages such as Arabic might contain sequences of words without spaces, etc.

2.5 Compositions and Optimizations

Because all NooJ linguistic resources are now compiled into a single type of finite-state machines, it is possible to merge and compose them. There are several remaining issues though:

– Priorities: each NooJ dictionary is associated with a level of priority that allows users to impose or filter out lexical solutions. Because this priority system is known in advance (i.e. before parsing takes place), RA’s compiler can produce a single dictionary that merges all dictionaries, storing only higher-priority entries into account;

– Symbols: a typical NooJ grammar will contain lexical symbols such as <eat> (to represent all the morphological variants of *to eat*) and syntactic symbols such as <ADV> (to represent all Adverbs). These symbols must be matched by applying the corresponding RA machines. Therefore, before applying a given grammar to a text, RA must first compile the machines for each symbol that occurs in the grammar. This operation is equivalent to composing the finite-state transducer of a given grammar with the finite-state transducers that correspond to each of its symbols.

3 Conclusion

In this paper, we have shown that it is possible to build a system that allows linguists to construct regular, context-free and context-sensitive grammars in a simple and unified way, and yet process these grammars using finite-state machines. The idea is to "flatten" Context-Free Grammars, and to use variables and constraints after the matching process to implement Context-Sensitive Grammars.

Having one unified machine for these three types of grammars allows us to compose and/or merge regular grammars (useful to describe morphology) with context-free grammars (useful to describe the structure of sentences) and context-sensitive grammars (useful to describe agreements and contextual constraints).

Moreover, this framework allows us to translate a complex series of cascading transducers into one single transducer. Although computing the final transducer is costly (exponential when flattening a recursive grammar), the resulting machine can then be applied to texts in $O(g\ t)$, which is ideal for industrial applications.

References

Chomsky, N.: Three models for description of language. In: IEEE (IRE) Transactions on Information Theory IT-2, pp. 113–124 (1956). Reprinted in Readings in Mathematical Psychology, vol. 2, pp. 105–124. Wiley, New York (1965)

Daciuk, J., Mihov, S., Watson, B.W., Watson, R.E.: Incremental construction of minimal acyclic finite-state automata. Comput. Linguist. **26**(1), 3–16 (2000)

Dalrymple, M., Kaplan, R., Maxwell, J., et al.: Formal Issues in Lexical-Functional Grammar. CSLI Publications, Stanford (1995)

Gazdar, G.: Applicability of indexed grammars to natural languages. In: Reyle, U., Rohrer, C. (eds.) Natural Language Parsing and Linguistic Theories. Studies in Linguistics and Philosophy, vol. 35, pp. 69–94. D. Reidel Publishing Company, Dordrecht (1988)

Kaplan, R., Bresnan, J.: Lexical-functional grammar: a formal system for grammatical representation. In: Bresnan, J. (ed.) The Mental Representation of Grammatical Relations, pp. 173–281. MIT Press, Cambridge (1982)

Kasami, T.: An efficient recognition and syntax-analysis algorithm for context-free languages. Technical report, AFCRL-65–758 (1965)

Linden, K., Silfverberg, M., Pirinen, T.: HFST tools for morphology: an efficient open-source package for construction of morphological analysers. University of Helsinki, Finland (2010)

Seljan, S., Vučković, K., Dovedan, Z.: Sentence representation in context-sensitive grammars. In: Suvremena lingvistika, vol. 53–54, pp. 205–218. Hrvatsko filološko društvo (2002)

Kuroda, S.-Y.: Classes of languages and linear-bounded automata. Inf. Control **7**(2), 207–223 (1964)

Penttonen, M.: One-sided and two sided context in formal grammars. Inf. Control **25**(4), 371–392 (1974)

Silberztein, M.: *Joe loves lea*: transformational analysis of direct transitive sentences. In: Okrut, T., Hetsevich, Y., Silberztein, M., Stanislavenka, H. (eds.) NooJ 2015. CCIS, vol. 607, pp. 55–65. Springer, Cham (2016a). https://doi.org/10.1007/978-3-319-42471-2_5

Silberztein, M.: Formalizing Natural Languages: The NooJ Approach. Wiley-ISTE, London (2016b)

Schmid, H.: A programming language for finite-state transducers. In: Proceedings of the 5th International Workshop on Finite State Methods in Natural Language Processing (FSMNLP), Helsinki, Finland (2005)

Karttunen, L., Tamás, G., Kempe, A.: Xerox finite-state tool, Technical report, Xerox Research Centre Europe (1997)