Preliminary Results from the Free Linguistic Environment Project

Abstract

We present ongoing work related to the Free Linguistic Environment (FLE) project, a grammar engineering platform for Lexical Functional Grammar (LFG) and related frameworks. In its present state, FLE is an environment that contains basic elements of a language processing pipeline, using morphological analysis and syntactic parsing with feature structures to generate parse-trees and other representations from input sentences. It can process CFGs and PCFGs fully, and targets full coverage of the complete set of XLE-grammar formalism by summer 2016.

1 Introduction

The Free Linguistic Environment (FLE) project aims at the development of a grammar engineering platform for Lexical Functional Grammar (LFG) (Bresnan, 2001; Dalrymple, 2001) and related frameworks. The goal is to create a platform-independent, open system that facilitates testing of algorithms and formal extensions of the current LFG-framework.

FLE currently contains a Probabilistic Context Free Grammar (PCFG) backbone that uses grammars that can be handcrafted or extracted and trained from common treebanks, independent lexical properties, feature specifications and related constraints. This is implemented atop a Weighted Finite State Transducer (WFST) (Mohri, 2004) that allows for extended probabilistic models to be applied to the transitions via weights, weight functions, or objects that encapsulate weight functions. The weight functions can entail Unification, or even Monotonicity Calculus computations (Icard III and Moss, 2014).

The project is motivated by a variety of concerns. One is to experiment with new algorithms within the LFG-framework that can facilitate probabilistic modeling research by involving Probabilistic Context Free Grammar (PCFG)s and using WFST models for the parser, among other approaches suggested in Kaplan (1996) and elsewhere. The use of such a flexible framework has the potential to trigger subsequent developments that improve the grammar engineering interface, or the debugging and tuning of grammars.

The environment is coded in standard C++11 and C++14, utilizing exclusively open components, including the C++ Boost framework (Schling, 2011) and additional specialized libraries like Foma (Hulden, 2009), OpenFST (Allauzen et al., 2007), and Ucto (Jain et al., 2012). Code generation is handled by the Backus-Naur Form Converter (BNFC) (Forsberg and Ranta, 2004) and the freely available lexer and parser generators flex and bison (Levine, 2009). The development environment requires CMake[1] and common working C++ compilers with at least C++11 support.

It is released under the Apache 2.0 license, as are most of the components that it uses.[2]

The resulting code-base is tested to compile on common operating system platforms, e.g. Windows, Mac OS X, and various Linux distributions. The binaries will provide libraries and executables for the common operating systems and linked modules for some programming languages, e.g. Python.

Besides providing an environment to test different algorithms and approaches to parsing natural language sentences with LFG-grammars, one purpose of the environment is to create a grammar a engineering platform or components that integrate better in common operating systems and computing environments. This includes not only Windows, Mac, and Linux platforms, but also virtual machines and distributed environments.

For various language documentation projects, in particular work on under-resourced and endangered languages, we need a platform that is not only embedded much better in current computing environments (including tablets and mobile thin-computers), but also one that is easy to use for grammar engineers. One of the major issues working with large morphologies or grammars is the identification of errors in failed parses. A central goal of this project is to provide a simple interface for grammar writers while allowing access to deep properties of the parsing steps and internal operations.

While we see a need for a parser and grammar engi-

[1]See `http://cmake.org/`.

[2]See `http://www.apache.org/licenses/LICENSE-2.0`. The Apache license appears to be more adequate for collaborative academic and industry projects. The one exception at the moment is the optional Ucto unicode tokenizer library, which is released under GPL version 3.0.

neering environment that provides help to documentary linguists and grammar writers, we also see a need for efficient implementations that are scalable, parallelized and distributed. By providing a library of atomic functions, we hope to create an architecture that we and others can subsequently optimize with respect to these goals.

The environment should also enable us to experiment with probabilistic models and extensions to the classical LFG-framework. Probabilistic LFG models would allow us to extend the spectrum of application in NLP and HLT, to address new research questions, and to boost grammar development and engineering using machine learning strategies and treebanks.

Our ultimate goal is to be able to integrate semantic components and processing in the parser to be able to make use of various kinds of inferencing and processing of other semantic properties.

2 General Architecture

Currently there is one experimental setting and implementation of FLE that uses a classical pipeline architecture for processing that consumes an input sentence, tokenizes it, and syntactically parses it on the basis of morphological analyses of the lexical items using different kinds of Chart parser implementations.

$$\begin{tikzcd}
\text{Input Sentence} \arrow[d] \\
\text{Tokenizer} \arrow[d] \\
\text{Morphological Analyzer} \arrow[d] \\
\text{Syntactic Parser} \arrow[d] \\
\text{c-structure \& f-structure}
\end{tikzcd}$$

Figure 1: Pipeline Architecture

Our goal is to provide the functionalities in this pipeline as library functions that can be arranged in a pipeline architecture (Figure 1) or in a parallel fashion using, for example, a blackboard-architecture (Jackendoff, 2007).

In the following we will briefly describe the current state of integration of necessary components in the processing chain.

2.1 Tokenization

FLE can process tokenized input and it provides a set of different tokenization approaches that can be integrated in the processing chain in various ways:

- **C++ Tokenizer subclass directly compiled into FLE.** The present system falls back to a simple whitespace tokenizer should no other be provided.
- **Foma-based Finite State Tokenizer.** A tokenizer that makes use of the Foma library (Hulden, 2009).
- **Ucto-based tokenizers.** Tokenizers that use Ucto (Jain et al., 2012)

At the time of writing we have developed a finite state tokenizer for Burmese (mya) – a Tibeto-Burman language written in *abugida* scripts with no spaces between words – using a wordlist and Foma regular expressions. We anticipate that other Foma-based tokenizers will be available within the codebase for English (eng), German (deu), Croatian (hrv), Mandarin (cmn), Polish (pol) and other languages in the near future.

2.2 Morphological Analysis

The currently-implemented pipeline makes use of Foma-based Finite State Morphologies using Lexc and Foma regular expressions that are also compatible with the Xerox Finite State Toolkit (XFST) (Hulden, 2009; Beesley and Karttunen, 2003).

The FLE codebase makes a partial implementation of a Foma-based Burmese (mya) morphological analyzer available, as well as a larger, integrated open English (eng) morphology. From here, work is planned on further morphologies for various under-resourced and endangered languages. Additionally, we will extend the support to other binary formats and formalisms, including OpenFST and the Stuttgart Finite State Toolbox (SFST) (Schmid, 2005).

2.3 Parsing

The architecture of the parser depends in part on the particular grammar formalisms, and the parsing strategy and the grammar properties determine the computational grammar representation. For example, in a left-corner parser, it would be expedient to use a representation that made access to the left-peripheral symbol of the right-hand CFG-rule efficient. Additionally, some grammar formalisms have properties that lend themselves to particular computational representations, e.g. using Finite State Machines. Our own implementations of CFG-formalisms are a case in point, making use of regular expression operators such as ⋆, +, ?, |, and the grouping brackets " (" and ") " to simplify the rule sets.

The different grammar formalisms are parsed and mapped on one internal grammar representation using WFSTs.

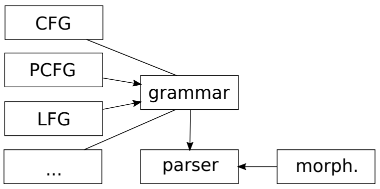

Figure 2: Grammar processing and parsing

This internal representation is based on the OpenFST library and can benefit from its extended capacities and features. It provides the functionality to represent different grammar formalisms in similar data-structures, rendering them compatible with a variety of parsing algorithms. The mapped grammars can be stored as binary files and exported in the Graphviz DOT format.

Each of the currently-supported grammar formalisms is formally defined using the Labeled Backus-Naur Form (Forsberg and Ranta, 2004), an extension of the common BNF. The BNF Converter (BNF, 2016) is used to generate the C++ code using intermediate Flex- and Bison- based lexer and parser generation (Levine, 2009). This generated code is then extended with the semantics to map grammars to internal WFST representations.

At the time of writing, there are parsing implementations for CFGs and PCFGs. These are optimized variants of the Earley Parser (Earley, 1968, 1970) which take into consideration several points discussed in (Aycock and Horspool, 2002), among others.

The implementation of Unification and feature logic handling currently under development uses *weights* in OpenFST WSFT instances as objects and functions. Our goal is to map the algorithm for uncertainty, the unification algorithm and other proposals discussed in Kaplan and Maxwell (1988); Maxwell III and Kaplan (1996, 1991); Maxwell and Kaplan (1993) to a WFST architecture in a systematic way.

3 Development Plan

Our current development plan includes these priorities:

1. full compatibility with the current XLE-environment

2. integration of a graphical environment for interaction

3. development of a Python-module interacting with the library and FLE-components

4. integration of initial semantic components (e.g. Monotonicity Calculus, Glue Semantics)

5. integration of a parallelized processing chain with a blackboard architecture

6. extension of the morphologies and grammars or grammar fragments to more languages

We expect to reach the first two goals in Spring 2016, and the Python module and the implementation of algorithms associated with the Monotonicity Calculus will be available as earyly as Summer 2016.

4 Conclusion

The current code and architecture is experimental and very much in flux. Nevertheless, we expect to have more than a simple test-setting of the initial pipeline ready for demonstration and testing by July 2016, in time for the conference.

Many of the currently implemented components might be useful to other projects. For example:

- The LBNF-specifications of grammar formalisms and BNFC can be used to generate parsers for the grammars in various other programming languages, including Java, Haskell, C#, and Python.

We have performed preliminary performance tests using various Foma-morphologies and the first parser implementation without Unification and feature logic. Currently the morphology can process more than 100,000 tokens per second on an Intel Core i7 PC with a bleeding edge Linux distribution and gcc/g++ 5.x. This includes only covered vocabulary with lexical ambiguities. The syntactic parser tested on a small grammar with structural and lexical ambiguities parses approx. 3,000 sentences per second with an average sentence length of 7 words. This suggests that an improved version using a WFST-based CFG-backbone can be expected to perform even better, at least prior to the inclusion of Unification and other weight functions.

Acknowledgments

We are grateful to many people for their help. For the purpose of anonymous reviewing, we remove the details here.

References

The BNF Converter, Feb 2016. URL `http://bnfc.digitalgrammars.com/`.

Cyril Allauzen, Michael Riley, Johan Schalkwyk, Wojciech Skut, and Mehryar Mohri. OpenFst: A General and Efficient Weighted Finite-State Transducer Library. In *Proceedings of the Twelfth International Conference on Implementation and Application of Automata, (CIAA 2007)*, volume 4783 of *Lecture Notes in Computer Science*, pages 11–23, Prague, Czech Republic, 2007. Springer.

John Aycock and R. Nigel Horspool. Practical Earley parsing. *The Computer Journal*, 45:620—630, 2002. doi: doi:10.1093/comjnl/45.6.620.

Kenneth R. Beesley and Lauri Karttunen. *Finite State Morphology*. CSLI Publications, 2003. URL `http://www.fsmbook.com`.

Joan Bresnan. *Lexical-Functional Syntax*. Blackwell, 2001. ISBN 0-631-20973-5.

Mary Dalrymple. *Lexical Functional Grammar*. Number 42 in Syntax and Semantics. Academic Press, New York, 2001. ISBN 0-12-613534-7.

Jay Earley. *An Efficient Context-Free Parsing Algorithm*. PhD thesis, Carnegie-Mellon University, 1968.

Jay Earley. An efficient context-free parsing algorithm. *Commun. ACM*, 13(2):94–102, February 1970. ISSN 0001-0782. doi: 10.1145/362007.362035. URL `http://doi.acm.org/10.1145/362007.362035`.

Markus Forsberg and Aarne Ranta. BNF Converter. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*, Haskell '04, pages 94–95, New York, NY, USA, 2004. ACM. doi: 10.1145/1017472.1017475.

Mans Hulden. Foma: a finite-state compiler and library. In *Proceedings of the 12th Conference of the European Chapter of the Association for Computational Linguistics*, pages 29–32. Association for Computational Linguistics, 2009.

Thomas Icard III and Lawrence Moss. Recent progress in monotonicity. *Perspectives on Semantic Representations for Textual Inference, LiLT (Linguistic Issues in Language Technology)*, 9, 2014.

Ray Jackendoff. A parallel architecture perspective on language processing. *Brain Research*, pages 2–22, 2007.

Anil K. Jain, Lin Hong, and Sharath Pankanti. Ucto: Unicode Tokenizer version 0.5.3 Reference Guide. Technical Report ILK 12-05, Induction of Linguistic Knowledge Research Group, Tilburg Centre for Cognition and Communication, Tilburg University, Tilburg, The Netherlands, November 2012. URL `https://ilk.uvt.nl/ucto/ucto_manual.pdf`.

Ronald Kaplan. A probabilistic approach to lexical-functional grammar, August 1996. Presentation at the LFG Colloquium and Workshops, Rank Xerox Research Centre.

Ronald M Kaplan and John T Maxwell. An algorithm for functional uncertainty. In *Proceedings of the 12th conference on Computational linguistics-Volume 1*, pages 297–302, 1988.

John R. Levine. *flex & bison*. O'Reilly Media, Sebastopol, CA, 2009.

John T Maxwell and Ronald M Kaplan. The interface between phrasal and functional constraints. *Computational Linguistics*, 19(4):571–590, 1993.

John T Maxwell III and Ronald M Kaplan. *Current Issues in Parsing Technology*, chapter A Method for Disjunctive Constraint Satisfaction, pages 173–190. 1991.

John T. Maxwell III and Ronald M. Kaplan. Unification parser that automatically take advantage of context freeness. In *Proceedings of the first LFG Conference (Grenoble)*, Stanford, 1996. CSLI Publications.

Mehryar Mohri. Weighted finite-state transducer algorithms. an overview. In *Formal Languages and Applications*, pages 551–563. Springer, 2004.

Boris Schling. *The Boost C++ Libraries*. XML Press, 2011. ISBN 9780982219195.

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