Abstract

The Free Linguistic Environment (FLE) project focuses on the development of an open and free library of natural language processing functions and a grammar engineering platform for Lexical Functional Grammar (LFG) and related grammar frameworks. In its present state the code-base of FLE contains basic essential elements for LFG-parsing. It uses finite-state-based morphological analyzers and syntactic unification parsers to generate parse-trees and related functional representations for input sentences based on a grammar. It can process a variety of grammar formalisms, which can be used independently or serve as backbones for the LFG parser. Among the supported formalisms are Context-free Grammars (CFG), Probabilistic Context-free Grammars (PCFG), and all formal grammar components of the XLE-grammar formalism. The current implementation of the LFG-parser includes the possibility to use a PCFG backbone to model probabilistic c-structures. It also includes f-structure representations that allow for the specification or calculation of probabilities for complete f-structure representations, as well as for sub-paths in f-structure trees. Given these design features, FLE enables various forms of probabilistic modeling of c-structures and f-structures for input or output sentences that go beyond the capabilities of other technologies based on the LFG framework.

1 Introduction

Our main motivation to launch the FLE project was to get access to a free and open parser environment for research and educational purposes. It aims to develop a grammar engineering platform for Lexical-Functional Grammar (LFG) (Kaplan & Bresnan, 1982; Bresnan, 2001; Dalrymple, 2001) and related grammar frameworks. The long-term goal is to create an open and platform-independent system that facilitates the testing of algorithms and formal extensions of the LFG framework. Among others, our interests are to merge current technologies and approaches in Natural Language Processing (NLP) with an LFG parser environment, and to integrate semantic analysis in the resulting computational environment.

The FLE project is motivated by a variety of concerns. One is to experiment with new algorithms within the LFG framework that can facilitate probabilistic modeling, as suggested in Kaplan (1996) and elsewhere.[1] The FLE environment should make it possible to experiment with probabilistic models and extensions to the classical LFG framework, as described below. Consequently, 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.

[†]We are grateful to Ron Kaplan, Ken Beesley, Lionel Clément, Larry Moss, Mary Dalrymple, Agnieszka Patujek, Adam Przepiórkowski, Paul Meurer, Helge Dyvik, Annie Zaennen, Valeria de Paiva for many helpful suggestions, data sets, grammar samples, ideas, and comments.

[1]While a reviewer pointed out that there are many approaches to quantitative LFG, to our knowledge there is no parser platform that implements probabilistic c-structure and f-structure generation or processing.

Besides providing an environment to test different algorithms and approaches to parsing natural language sentences with LFG-based grammars, one purpose of the project is to create a grammar engineering platform that integrates better in common operating systems and computing environments.

For language documentation projects, in particular work on under-resourced and endangered languages, we need a platform that is not only usable on tablets and mobile thin-computers like Chromebooks, but also one that is easy to use for grammar engineers without strong technological skills.

While we see a need for a parser and grammar engineering environment that provide help to documentary linguists and grammar writers, we also see a need for efficient implementations that are scalable, parallelized and distributed. This might have been a major concern by many parser developers in the past. Given the ongoing changes in computing hardware, infrastructure, and environments, it is a permanent challenge to constantly adapt algorithms and code to be able to benefit from ongoing innovations. 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.

Our ultimate goal is to integrate semantic processing and computational components in some future version of the library and the resulting parsers.

2 Existing LFG Grammar Engineering Platforms

There are various grammar engineering platforms on the market. In the following we will concentrate only on the LFG related platforms and component software environments that could benefit from those.

The Xerox Linguistic Environment (XLE) is the most significant and complete implementation of the LFG framework in a grammar engineering environment. It is accompanied by a large amount of documentation in form of textbook sections (e.g. Butt et al. (1999)), online documents (e.g. Crouch et al. (2011)), and academic publications (e.g. Maxwell & Kaplan (1996a)).

The FLE-project aims at reaching compatibility with XLE. XLE is not freely available. This prevents us from studying and teaching the particular algorithms related to LFG-parsing. The graphical environment of XLE has a rather vintage appearance with limited grammar engineering functionalities. It is available for Unix-based and – in an older version also – for Windows operating systems.

Another LFG-based grammar engineering environment is XLFG (Clément, 2016; Clément & Kinyon, 2001). It comes as a web-based platform that can be used in a browser window. It is not openly available and it is accompanied by only sparse documentation. We have been discussing the possibility of integrating the grammar formalism supported by XLFG in the FLE environment.

There is a growing number of environments that are used for the application of NLP components, for example, tokenizers, parsers, named entity recognizers, etc. However, none of these, to our knowledge, has integrated LFG parsers.

The Unstructured Information Management Architecture (UIMA) (Ferrucci & Lally, 2004), for example, is a component software architecture that allows speech and language processing engineers to define a processing pipeline of NLP components for the analysis of data sets (e.g. texts or even audio recordings), and the aggregation and visualization of analysis results. All FLE components can be used within an UIMA-based application, if the necessary wrappers for the UIMA infrastructure are provided.

The General Architecture for Text Engineering (GATE) (The GATE Team, 2011) is another such environment that is not geared towards the engineering of morphologies, LFG-based grammars, or specific NLP components, but rather towards the application of such components on some textual data. The integration of FLE components in the GATE environment is possible.

Existing educational and experimental environments for NLP, like the Natural Language Toolkit (NLTK) (Loper & Bird, 2002; Bird et al., 2009) provide various algorithms and tools implemented in the Python programming language. There is no LFG parser or adequate morphological analyzer integrated in NLTK yet. NLTK components are coded in Python and not necessarily tuned for efficiency and use with large data sets. The NLTK license also excludes commercial use and thus significantly differs from the FLE license. It is possible, though, to create an FLE-module for Python that uses the compiled C++ classes that are part of FLE.

3 Architecture

While the development of FLE focuses on providing a collection of algorithms necessary to read LFG formalisms and parse with them, we do implement experimental parser pipelines or fragments of those to test our algorithms. Currently, there is one experimental setting and implementation of FLE that uses a pipeline architecture for processing which 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:

$$\text{Input Sentence} \rightarrow \text{Tokenizer} \rightarrow \text{Morphological Analyzer} \rightarrow \text{Syntactic Parser} \rightarrow \text{c-structure \& f-structure}$$

As discussed in the next subsections, this is a very common way to arrange linguistic processing components in most common NLP architectures that involve raw sentence input processing and syntactic parsing. Our goal is not to provide a component architecture for linguistic processing modules, but to provide the library functionalities that can be arranged in different general architectures that enable LFG type parsing of raw sentence input.

The pipeline architecture above is common in Natural Language Processing applications. For a more cognitively or psycho-linguistically adequate model we can also arrange the language processing components in a parallel fashion, as proposed in Jackendoff (1997, 2007). In this kind of architecture the individual lan-

guage components generate representations that are mapped or synchronized with representations generated by other language processing components. The mapping constraints can be implemented as selection functions or constraints over possible representations. An implementation of such a parallel processing environment can be achieved using a blackboard or alternative message passing architecture.[2]

In the following sections, we will briefly describe the current state of integration of NLP components necessary for any kind of processing chain.

3.1 Grammar Formalisms

A prerequisite for any grammar-based parser is at least one grammar formalism that specifies the format for rules as supported by the parser. The grammar formalisms that FLE supports include a generic CFG-formalism, two different PCFG-formalisms, and the full set of XLE grammar components.

The CFG-formalism allows for the use of regular expression operators in the right-hand-side of rules, such as `*`, `+`, and `?`, round brackets for grouping of symbols, and curly brackets with the disjunction operator `|`, as in the following example:

```
S --> NP VP
NP --> {Art|Q} N
NP --> {Art+|Q*} N
VP --> ( Adv )* V
```

The use of regular expression operators is similar to the rule formalism used in XLE. The right-hand-sides of the rules are mapped on specific Finite State Machines in FLE that include cyclic paths or recursion, optionality, and disjunction. The XLE parser uses a similar CFG-backbone.

An extension of this formalism allows additional augmentation of rules with probabilities. This could be considered a version of a PCFG, in which the probability is associated with a complex right-hand side that may contain the regular expression operators as described above.[3]

```
1.0 S --> NP VP
0.7 NP --> {Art|Q} N
0.3 NP --> {Art+|Q*} N
1.0 VP --> ( Adv )* V
```

For compatibility reasons we have added the NLTK supported PCFG-formalism to FLE. This formalism uses CFG-rules in the Chomsky Normal Form (CNF). In this format, right-hand-sides of rules are augmented with their particular probability in square brackets. The right-hand-sides are optionally grouped to their corresponding left-hand-side symbol using the disjunction operator `|`.

[2] See Erman et al. (1980), Corkill (1991), or Hayes-Roth (1985) for some such solutions.

[3] Note that the interpretation of the probabilities in this formalism is open and can be used in many different ways. A discussion of some of the possibilities to interpret and use these probabilities would be beyond the scope of this paper.

```
S -> NP VP [1.0]
NP -> Det N [0.5] | NP PP [0.25] | 'John' [0.1]
NP -> 'I' [0.15]
Det -> 'the' [0.8] | 'my' [0.2]
N -> 'man' [0.5] | 'telescope' [0.5]
VP -> VP PP [0.1] | V NP [0.7] | V [0.2]
V -> 'ate' [0.35] | 'saw' [0.65]
PP -> P NP [1.0]
P -> 'with' [0.61] | 'under' [0.39]
```

As we will discuss in more detail, the inclusion of the two PCFG-type formalisms allows us to extend the back-end of FLE to include probabilities for c-structure representations in LFGs and the parsing algorithm.

3.1.1 XLE Grammar Formalism Parser

In addition to the existing grammar formalisms as described above, FLE provides parsers for all XLE-based grammar sub components. XLE grammars potentially consist of multiple specifications of sections that are labeled: CONFIG, FEATURES, LEXICON, MORPHOLOGY, MORPHTEXT, RULES, TEMPLATES. Each of these sections uses a specific formalism or language to specify configuration parameters for the parser, the features used in the lexicon, morphology, and syntax, or LFG rules. A description or explanation of all these sections is beyond the scope of this article. Please consult the XLE documentation online (Crouch et al., 2011) for more details. In the following sections, we will briefly discuss the parsers for the XLE RULES and FEATURES.

The following rules are taken from the toy-grammar that is part of the XLE documentation.

```
S --> e: (^ TENSE);
    (NP: (^ XCOMP* {OBJ|OBJ2})=!
         (^ TOPIC)=!)
     NP: (^ SUBJ)=!
         (! CASE)=NOM;
    { VP | VPaux }.
```

The following LBNF rules represent a segment of the LBNF-specification for the RULES section:

```
Grammar.        GRAMMAR ::= [RULE] ;
RuleS.          RULE    ::= LHS "-->" [RHS] RULEES ;
RuleS2.         RULE    ::= LHS "=" [RHS] RULEES ;
RuleEndSymbol. RULEES  ::= ".";
RHSSymbolOptional.    RHS ::= "(" RHSSYMBOL ")" ;
RHSymbolsDisjunction. RHS ::= "{" [ORHS] "}" ;
```

These arbitrarily selected LBNF rules display the specification of the `GRAMMAR` as a list of `RULE` symbols. Each rule is specified as a left-hand-side `LHS` followed

by a production symbol/string "=" or "->" and a list of right-hand-side symbols RHS. Each rule is terminated by a RULEES symbol, the rule-end symbol. Some examples for right-hand-side symbols are given in the final two rules, i.e. an optional symbol or a list of disjoint symbols.

As described in the previous section, FLE allows for different types of grammar backbones. It can be used with a CFG or PCFG backbone. The PCFG-capability allows for the use of grammars that can be handcrafted or extracted and trained from common treebanks. The internal encoding of the grammar is implemented atop a Weighted Finite State Transducer (WFST) (Berstel & Reutenauer, 1988; Kuich & Salomaa, 1986; Salomaa & Soittola, 1978).[4]

The WFST-architecture of the grammar backbone provides a simple way to not only encode a probabilistic backbone, but also to encode and process independent lexical properties, feature specifications and related constraints in a probabilistic manner. It allows for extended probabilistic models to be applied to the transitions via weights, weight functions, or objects that encapsulate complex weight functions. For example, the weight functions can entail unification or even semantic operations.

In FLE, the parsed XLE RULES section of a grammar is mapped onto a WFST as shown in Figure 1. The upper tape in the WFST in Figure 1 is represented by the initial symbol preceding the first "/", the lower tape corresponds to the embedded symbol between two "/". In this example, an initial probability or weight of 1 is assigned to all transitions, as represented by the final element following the final "/". These probabilities or weights can be changed, "trained", and subsequently tuned, if the weights are not already provided by the grammar. For example, the weights are provided when a PCFG-backbone is utilized.

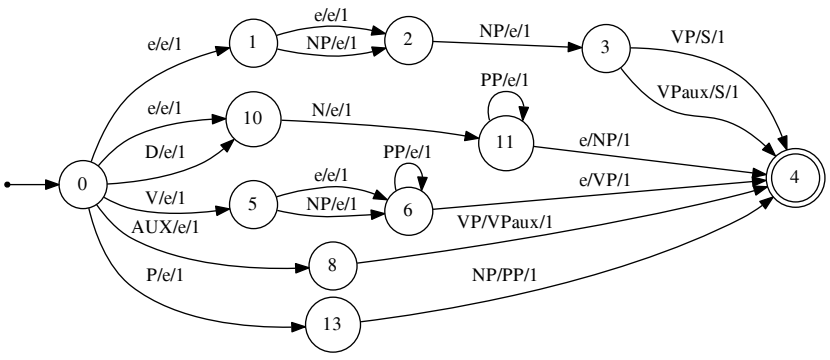

Figure 1: Mapping a CFG on a WFST

F-structures, as specified in the toy-grammar RULES section above, are not displayed in this WFST in Figure 1. These are left out for simplification reasons.

[4]See (Mohri, 2004) for an overview.

We discuss feature representations and unification in section 3.4 below.

Notice that the WFST in Figure 1 contains epsilon transition symbols on the upper tape, the lower tape, or on both tapes of the transducer. These are either specified in the grammar, as for example, in the sentence rule ("`S -> e (NP) NP { VP | VPaux }`") above, or they are inserted by the grammar-to-WFST mapping algorithm to express, among other properties, optionality.

Given the compact representation of the grammar rules as a WFST, we are able to model LFG grammars for bottom-up, top-down, and probabilistic parsing or generation. That is, the upper or lower tape can be used for parsing or generation. The upper tape paths represent the right-hand-sides and the lower-tape – the left-hand sides of the rules from a CFG-backbone. A top-down parser would use the lower tape. A bottom-up parser would use the upper tape to match input sequences or edges and then replace them with their counterpart on the opposite tape. The lower tape could be used to replace left-hand-side symbols with the corresponding right-hand-sides to generate output c-structures and sentences (assuming an existing f-structure and corresponding mapping functions).

In addition to the flexibility of this grammar representation for parsing and generation, we add the capability of probabilistic modeling of c-structures. The transitions in the WFST could be weighted with probabilities that correspond to rule probabilities in PCFGs. Alternatively, they could be used as pure transitional probabilities for symbols and paths without any PCFG association that may be learned from corpora or parser application. In this way we can represent the CFG-backbone of the LFG grammar as a probabilistic model. Simply put, if we map PCFG rule-probabilities on transition probabilities in the WFST, we can calculate the probability of a c-structure as the product of the probabilities of the rules used to generate or parse it. There are various ways in which the weights in a WFST can be used or specified to enable probabilistic modeling of c-structures in LFGs. We will discuss various options in subsequent work.

Note that the WFST-based model provides more possibilities and potential extensions to the grammar backbone. As discussed above, weights in the WFST could be used as transitional probabilities or rule probabilities mapped on transition probabilities to estimate the likelihood of c-structures. Weights could also be complex functions and data-structures that, for example, combine probabilities with f-structures and unification operations. At the same time, the CFG backbone can be extended by allowing the lower tape, i.e. the CFG-rule's left-hand-side to be extended. This data structure would allow for the formulation and use of context-sensitive rules by specifying contexts and even f-structures of left-hand-side symbols. We will have to postpone the discussion of such extensions to future work.

3.1.2 Formal Specification and Parser Generation

The grammar formalisms that FLE can read are specified using an extension of the Backus-Naur Form (BNF).[5] A BNF specification of a grammar contains a set of replacement rules. The rules in the sample below are interpreted as: a.) a `Grammar` consists of a list of `Rule` symbols, b.) a symbol surrounded by square brackets is interpreted as a list, c.) a `Rule` consists of a `LHS` (left-hand-side) symbol, d.) the production symbol `->` as a literal string, and e.) a list of `RHS` (right-hand-side) symbols. In our formal specification of grammar formalisms we used the the Labeled Backus-Naur Form (LBNF) formalism (Forsberg & Ranta, 2005). The LBNF formalism is used by the BNF Converter (BNFC) (Forsberg & Ranta, 2004) to generate parser code in a variety of programming languages. The labels here are prefixes to the rules, e.g. `Gram` and `Rul` separated from the rule by a "`.`". They are used for constructing a syntax tree that contains sub-trees that are defined by the non-terminals of the labeled rule.

```
Gram. Grammar ::= [Rule] ;
Rul.  Rule    ::= LHS "-->" [RHS] ;
```

The BNF Converter (BNFC) is independent software written in Haskell that generates code for the programming languages C, C++, Java and Haskell.[6] Many other languages can and surely will be added to the converter. The generated code constitutes a functioning syntactic parser for the grammar formalism that generates a parse tree for the grammar and rules in the specific formalism. These parse trees need to be mapped to data structures for processing in FLE. This is what we refer to as the semantics of the formalism parsers.

Since the FLE code-base is based on C and C++, we use the BNFC conversion to C++ for our grammar formalism parsers. The LBNF specification is converted to freely available lexer and parser generators *flex*[7] and *bison*.[8] The generated *flex* and *bison* code is wrapped in a C++ class using the Visitor Design Pattern.[9]

Specifying the grammar formalisms in LBNF and using BNFC for parser code generation has many advantages. The LBNF-specification of the formalisms for CFG, PCFG, and XLE grammar components provides an intuitive and easy understandable representation that allows for convenient adaptations, extensions, corrections and changes. The BNFC generated code and the Visitor Design Pattern that

[5] A discussion of the detailed specification of the different grammar formalisms that FLE supports would go beyond the scope of this paper. The formal specifications are available in the online Bitbucket repository of FLE at the following URL: https://bitbucket.org/dcavar/fle.

[6] BNFC also generates a LaTeXand PDF documentation of the formal language specification in a LBNF.

[7] As explained in Levine (2009), *flex* was written by Vern Paxson around 1987 in the programming language C. It generates code in C or C++ for lexical analyzers and tokenizers for parsers of formal languages that are based on Deterministic Finite Automata (DFA).

[8] As explained in Levine (2009), *bison* is a parser generator that was originally written by Robert Corbett in 1988. Given a CFG-specification for a formal language it generates parser code in the programming languages C, C++, or Java.

[9] See Gamma et al. (1995) for more details).

we utilized provide parser code in various programming languages and at the same time minimize the coding effort for the implementation of mappings to internal data structures and representations needed by different parsing algorithms. Given the openness and free licensing of all the necessary components, we contribute to the sustainability of the grammars that are based on the supported formalisms, including the XLE grammars.

3.2 Tokenizer

FLE can process tokenized input. It also provides a set of different tokenization approaches that can be integrated in the processing chain in various ways. A tokenizer provides a list of *tokens* for a given input sentence. The sentence *John reads a book.* is tokenized into the sequence of strings `["John", "reads", "a", "book", "."]`. The following tokenization strategies and components are implemented: a.) **C++ tokenizer subclass directly compiled into FLE.** The present system falls back to a simple whitespace tokenizer, should no other be provided; b.) **Foma-based Finite State tokenizer.** A tokenizer that makes use of the Foma library (Hulden, 2009); c.) **Conditional Random Field (CRF).** A tokenizer based upon CRF machine learning (see, for example, Wallach (2004)) which does segmentation for Chinese; d.) **Ucto-based tokenizers.** Tokenizers that use Ucto (Jain et al., 2012).

At this time, we are working on tokenizers for languages such as Burmese (mya) and Chinese (zho). We have developed FST-based tokenizers for Burmese (mya), a Tibeto-Burman language written in *abugida* scripts with no spaces indicating word boundaries, and for Chinese (zho). These tokenizers use wordlists and Foma regular expressions, providing a baseline left-to-right maximum segmentation for Burmese or Chinese text. For both Burmese and Chinese, we are experimenting with improved segmenting algorithms using Conditional Random Field (CRF) sequence tagging (Tseng et al., 2005), as described below.

Other Foma, Ucto, and CRF-based tokenizers will be available in the codebase for English (eng), German (deu), Croatian (hrv), Mandarin (cmn), Polish (pol) and other languages in the near future.

3.2.1 Segmentation of Chinese and Burmese

Unlike English and many other Indo-European languages, Asian languages, such as Chinese and Burmese, do not mark word boundaries with spaces. Thus, to parse Chinese and Burmese in FLE, we first need to segment the sentences into words so that they can be passed into the morphological analyzer (or even part-of-speech tagger) and subsequently to the parser.

In FLE, we use CRF for Chinese segmentation, because this method has achieved the best performance previously, and the dictionary-based longest match algorithm for Burmese because it has yielded competitive results using only very limited

language resources. We will briefly discuss two approaches for tokenization for Chinese and Burmese.

Segmentation of Chinese Using Conditional Random Field Essentially, the segmentation task is seen as a sequence labeling problem, where the CRF model learns to give labels to each character as to whether it is a word boundary or not. To implement a CRF segmenter, we used the C++ library *dlib* (King, 2009).

As of now, our CRF segmenter uses three types of features as described in Zhao et al. (2010), i.e. character unigram features (previous character C_{-1}, current character C_0 and next character C_{+1}), character bigram features ($C_{-1}C_0$, C_0C_{+1}, $C_{-1}C_{+1}$) and character type features. The character type features denote which of the five types the previous, current and next character belongs to. We follow Zhao et al. (2010) to classify characters in Chinese into five types: numbers, characters referring to time (year, month, day, etc.), English letters, punctuation and other Chinese characters. The intuition is that the character type information of the neighboring characters will be helpful for the CRF segmenter to decide on word boundaries.

We used training data from the Second International Chinese Word Segmentation Bakeoff (Emerson, 2005) to train our model. Our current result for the test data of the bakeoff is **93.0%** on recall, **94.1%** on precision and an F score of **93.6%**, using the official scorer from the bakeoff. In the next step, we will further tune the parameters and experiment with other types of features to improve our results.

Segmentation of Burmese using Foma Our goal is a language independent system for low-resourced languages using openly available resources without relying on tagged corpora. We implemented a Finite State Machine for the longest match algorithm (Poowarawan, 1986) using Foma (Hulden, 2009) and a word list for Burmese (LeRoy Benjamin Sharon, 2016).

For an experiment, we tested the segmenter on a Wikipedia article with **3302** words in **77** sentences. The sentences were hand-segmented by a native Burmese speaker. The baseline performance of the dictionary-based segmentation gave **86.92%** precision and **92%** recall. We plan to integrate this with the CRF machine learning approach with morphology as discussed above for Chinese.

3.3 Morphology

In the current codebase of FLE we make use of Foma-based Finite State Transducer (FST) morphologies using Lexc and Foma regular expressions that are also compatible with the Xerox Finite State Toolkit (XFST) (Hulden, 2009; Beesley & Karttunen, 2003).

The FLE codebase includes an open English (eng) morphology. This morphology currently contains the all irregular verbs and nouns, most of the closed class lexicon, a large set of open class items, including a broad variety of named entities (e.g. toponyms, anthroponyms, and institutions and companies). We are also able to process multi-word expressions and unknown morphemes efficiently using

a multi-word recognizer and unknown word guesser that wraps the Foma-based morphology. The guessing algorithms that we experiment with in the morphology includes lexical category and morphological feature guessing using Hidden Markov Models (HMMs). These can be trained to guess the detailed features required by the LFG parser. An integration of the guesser into the syntactic parser will be done in a future development phase. A detailed description of this approach to handle unknown words is beyond the scope of this paper.

In its current version, the English morphology contains 117,705 paths, which corresponds to the number of surface word forms with ambiguities. We estimate that it contains 38,964 morphemes, including inflectional and derivational suffixes and prefixes. The binary form of the FST is 1.9 MB large, with 88,783 states and 124,221 arcs. These numbers are expected to increase significantly before the final public release of the source and the binaries.[10]

The following example illustrates the output from our multi-word enabled English morphology for an input sentence like *Tim Cook, the CEO of Apple, works now for Google*.

```
0 1 Tim Tim+N+Sg+Masc+NEPersonName
1 2 Cook Cook+N+Sg+NEFamilyName
3 4 the the+D+Art+Def
4 5 CEO CEO+N+Sg+Abbrev
5 6 of of+P
6 7 Apple Apple+N+NEBusiness
8 9 works work+N+Pl, work+V+3P+Sg
9 10 now now+Adv+Temporal
10 11 for for+P, for+P+Time
11 12 Google Google+N+NEBusiness
0 2 Tim Cook Tim Cook+N+Masc+NEPersonName+NESTBusiness
```

The tab-delimited fields in the output consist of the index of the token span for the respective token sequence (that is from and to token positions), followed by the token sequence itself and a list of comma separated potential analyses. We extend common feature sets with named entities that include `NEPersonName`, `NEFamilyName`, `NEBusiness` (a business name), or `NESTBusiness` (a subtype of any named entity that is business related). We will be able to expand abbreviations like `CEO` to their full forms in the final release of the complete morphology.

Most of the lexical category and feature labels do not yet conform to any standard, nor are they synchronized with proposals from other relevant projects or best-practice recommendations. In the final release of the morphology we will take standardization into account and very likely change the category and feature labels of the existing morphologies accordingly.

Within the FLE environment we also provide partial implementations of Foma-based morphologies for other languages, including Burmese (mya), Mandarin Chi-

[10]The size increase would not affect the size of the binary FST-file in a significant way, rather the number of paths and covered morphemes and morphotactic regularities.

nese (cmn),[11] or Croatian (hrv).[12] 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.

3.4 Features and Unification

F-structures in FLE are represented as Directed Acyclic Graphs (DAG). The Attribute Value Matrix in Figure 2 is a simplified structure representing basic morpho-syntactic features.

$$\begin{bmatrix} \text{AGR} & \begin{bmatrix} \text{NUM} & \text{PL} \\ \text{GEN} & \text{M} \end{bmatrix} \\ \text{TNS} & \text{PAST} \\ \text{ASP} & \text{PERF} \end{bmatrix}$$

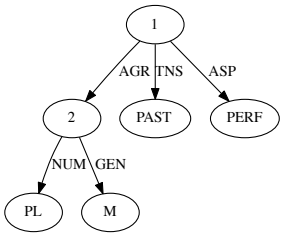

Figure 2: F-structure as Attribute Value Matrix (AVM) and DAG

In FLE, these AVMs are mapped to DAGs as in Figure 2. Note that, although the DAG shows the terminal values as states, the internal representation of DAGs that we use represents them as edges.

The graph is defined as a set G of edges that are represented as tuples $\{f_i, t_i, s_i, w_i\}$, where f is the start-state of the edge, t the target state, s the associated symbol (an attribute or value), and w a weight. In our implementation a DAG is a set G of such weighted edges that is associated with a general weight p for the entire DAG. This implementation of a f-structure allows us to associate weights with edges or entire paths in the DAG. It also allows us to represent the probability of a specific DAG and thus the encoded f-structure independently in the DAG weight. Such a weight could be estimated using a corpus of c-structures and f-structures, or applying parses to a corpus of sentences.

This implementation of f-structures allows us to model associations between f-structures and c-structures based on probabilities. A probability of a sentence could be described as the conditional probability $P(s_i) = P(f_n \mid c_n)$, where c is one possible c-structure n for the sentence s_i, and f is the corresponding f-structure for c. Probabilistic feature representations provide new possibilities for

[11] As one reviewer correctly points out, Mandarin Chinese is not a morphologically rich language, thus the morphology should be rather called a lexicon or FST-based word-list. This assessment is true, if we focus only on the *Han character*-based orthography. Word formation in Mandarin Chinese could indeed be described in terms of affixation and morphotactics, as described in Packard (2000). If we consider the Romanized orthography using Pinyin, it indeed appears plausible to describe word formation using a two-level morphology based approach.

[12] A Croatian morphological analyzer was compiled using a very different FST-based framework (Cavar et al., 2008). We are transferring the morphology to a Foma-based lexc and regular expression format.

robust unification or adaptation to speaker or domain specific feature properties.

Our graph-based implementation of unification is based on two specific constraints. All DAGs are stored in a uniform DAG-space. All edges that represent attributes and values in those DAGs are mapped uniquely. A synchronization object replaces all symbols (attribute names and concrete values) with unique numerical IDs across the entire DAG-space. All states and paths receive a uniform reference, i.e. in all DAGs an edge representing the Attribute-Value path `ROOT -> AGR -> NUM -> PL` will be mapped onto a uniform edge of numerical reference points `(1, 18, 21, 35)`, with 1 representing $ROOT$, 18 representing AGR, and so on.[13] A path is a set G of edges $\{f_i, t_i, s_i, w_i\}$. Terminal edges in G are specific. Their t-value is always 0. Our unification algorithm creates the union of all edges in two sets G_1 and G_2 and it only places a matching constraint on terminal-edges, i.e. unification fails if for two edges $f_1 = f_2$, $t_1 = t_2 = 0$, and $s_1 \neq s_2$. In general, two DAGs are unified by copying one and building the edge union of the copy with the other. Copying is necessary because unification might fail and the previous DAG/f-structure of the first is needed in subsequent unification steps or attempts.

Although this unification algorithm is very efficient, more complicated filters and conditions are necessary and will be implemented in future. In the current version it is sufficient for initial experiments. Future developments might result in revisions and changes in the data structures and related algorithms.

3.5 Parsing

The architecture of the parser depends partly on the particular grammar formalism that is used but also on the underlying linguistic components. The parsing strategy and the grammar properties determine the computational grammar representation and particular parsing algorithms. For example, using an Earley parsing algorithm (Earley, 1968) requires a specific mapping of a CFG-grammar to data structures that makes use of left-peripheral symbols of the right-hand-side of rules. Mapping a grammar to a Finite State Machine (FST) with or without a stack, using the OpenGrm Thrax environment, for example(Tai et al., 2011), requires a very different parsing algorithm and grammar data structure internally.

We have implemented a CFG and a PCFG formalism parser, the possible backbones for our LFG-grammars, using the LBNF formalism. These implementations make use of regular expression operators in the right-hand-side of rules, such as `*`, `+`, and `?`. We allow for the use of `|` as a disjunction operator, and grouping brackets "`(`" and "`)`" to simplify the rule sets. This is similar to the rule formalism used in XLE. Finite State Machines are ideal for mapping right-hand-sides of rules with such properties (e.g. recursion, optionality, and disjunction).

In one implementation of the mapping of grammars to parser-internal data structures, we make use of Finite State Transducers (FST), such that the one side

[13] The assignment of numerical values to symbols happens during run-time when parsing the grammar.

of CFG rules is read and the other written. That is, the left-hand-side of a CFG rule is emitted when a certain right-hand-side is processed.

As mentioned above, to be able to integrate probabilities in the grammar representation, we use a WFST (Allauzen et al., 2007). This allows us not only to store complex right-hand-sides of rules in an FST data-structure, but also to represent rule probabilities of a PCFG-type in the same data structure.

In addition to the mentioned CFG and PCFG formalisms, we have implemented an LBNF specification of grammar rules used in XLE. For core rule types of the XLE grammar formalism, we have also integrated the semantics, i.e. the mapping of rules to internal data-structures or functions in the parser in the following way.

The different grammar formalisms are parsed and mapped on one internal grammar representation using a WFST. This internal representation is based on our own implementation of a WFST. We intend to base future versions on the the OpenFST library. Our own implementation is simpler and more efficient than the OpenFST version, and initially we did not foresee a need for functions like minimization or ϵ reduction. Avoiding dependencies on external libraries and tools is our core strategy. An integration of OpenFST, however, would save us implementation of various FST-based operations and WFST features that are already integrated in it.

Our use of a WFST as a data structure for the parsing algorithm allows different grammar formalisms to be represented in similar data structures, rendering them compatible with a variety of parsing algorithms. The mapped grammars can also be stored persistently as binary files and exported, for example, to various other FST exchange or visualization formats.

In the previous section we have demonstrated how the grammar formalisms are mapped onto WFST representations internally. We have also mentioned that each of the current grammar formalisms is defined using the LBNF formalisms (Forsberg & Ranta, 2004), an extension of the common BNF formalism. The BNF Converter (Forsberg & Ranta, 2004) is used to generate the C++ code implementation of the syntactic parsers for these formalisms using an intermediate *flex*-based and *bison*-based code for lexer and parser generation (Levine, 2009) underlyingly. This generated code is extended with the semantics to map grammars to internal WFST representations. The effort of porting the grammar representations to other internal formats or even other programming languages is thus significantly reduced.

Note that the mapping of CFGs to WFSTs does not include a reduction of expressiveness of the underlying CFG formalism. While we do intend to experiment with a reduction of CFGs to FSTs by limiting the recursion depth of specific recursions, in this case the WFST is applied recursively by our parser without any effect on the complexity of the grammar as such.

The parsers for the (P)CFG backbone and the XLE RULES specification are implemented mapping to WFSTs. The parsing algorithms that we test include an optimized variant of the Earley Parser (Earley, 1968, 1970). In the implementation, we took into consideration several implementation and optimization suggestions discussed in (Aycock & Horspool, 2002), among others. The specific implementa-

tion of the parsing algorithm makes use of the WFST-encoded grammar. It recursively generates edges from the WFST representation based on the input tokens in a bottom-up fashion. Instead of positions in dotted rules, the edges are tuples that point to the span of input tokens and to a state in the WFST.

While we use feature representations in form of DAGs and a unification algorithm that is based on such DAGs (see for example Shieber (1985)), as described in the previous section, our ongoing evaluation is concerned with the question of applying unification after the possible c-structure representations are generated, or during the generation of edges while tracking paths through the WFST. In the implementation of WFSTs *weights* can be implemented as instances of objects[14] or functions. This allows us to associate not only a probabilistic metric to the transitions through the WFST, but also operations like unification and resulting f-structures.

Our goal is to integrate the algorithm for uncertainty, the unification algorithm, and other proposals discussed in Kaplan & Maxwell (1988); Maxwell & Kaplan (1996a,b, 1991, 1993) into the WFST architecture in a systematic way. We are able to handle probabilities in the c-structure representations in the form of weights in the WFST that could be mapped from a PCFG backbone or quantification of rule applications while processing corpora. As mentioned above, these probabilities can also be related to f-structures using DAG probabilities or products of path probabilities in the resulting DAGs. We have not performed any experiments related to c-structure and f-structure probabilities.

4 Development Plan

The FLE codebase and the entire environment with external libraries are coded in standard C++11 and C++14. They utilize exclusively open components, including the C++ Boost framework (Schling, 2011) and additional specialized libraries like Foma (Hulden, 2009), OpenFST (Allauzen et al., 2007), OpenGram Thrax (Roark et al., 2012), Ucto (Jain et al., 2012), and Dlib-ml (King, 2009). The parsers for formalisms in the FLE library are implemented using the Labeled Backus-Naur Form (LBNF). The syntactic parser code for these grammar formalisms is generated by the Backus-Naur Form Converter (BNFC) (Forsberg & Ranta, 2004), and for C and C++ the freely available lexer and parser generators *flex* and *bison* (see Levine (2009) and footnotes 7 and 8). The development environment requires CMake[15] and common working C++ compilers with at least C++11 support.

The entire FLE environment is released under the Apache License 2.0, as are most of the components that it uses.[16] This includes all code, morpheme collections and trained models that we created ourselves and that we legally can distribute. The Apache License appears to be more adequate to facilitate collaborative

[14] Any C++ object for example could function as a weight.

[15] See http://cmake.org/.

[16] See http://www.apache.org/licenses/LICENSE-2.0.

academic and industry projects. It allows one to freely download and use the FLE code or binaries for personal, company internal, or commercial purposes. The system can be integrated in third party systems and packages. Any modifications that are made to the FLE code do not have to be shared and redistributed, although this is encouraged. The license requires that any reused and distributed piece of the FLE source code or binary software has to contain proper attribution and that a copy of the license has to be included in any redistribution.

Most of the components and external libraries that are used in FLE are released under the Apache License 2.0. The one exception at the moment is the optional Ucto Unicode tokenizer library, which is released under GPL version 3.0.

The resulting codebase 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.

Our development plan includes some of the following goals, without implying any priorities: 1.) Full compatibility with the grammars of the current XLE environment. While we finalized parsers for the different grammar sections of XLE grammars, not all features of the grammar have been semantically mapped onto internal data representations or functions; 2.) Integration of a graphical environment for the parser and grammar development; 3.) Development of a Python module interacting with the library and FLE components; 4.) Integration of initial semantic components (e.g. Glue Semantics); 5.) Integration of a parallelized processing chain with a blackboard architecture; 6.) Extension of the morphologies and grammars or grammar fragments to other languages than English, Burmese, Chinese, Croatian etc., by integrating specific tokenizers and morphological analyzers; 7.) Since we foresee the possibility for generating initial PCFGs from treebanks to bootstrap the grammar engineering process, we need to map tags from treebanks to morphological feature annotations and LFG-type feature representations.

To integrate and experiment with a PCFG backbone we developed initial PCFG extraction tools that can process treebanks of the Penn Treebank format (Marcus et al., 1993). For languages that have existing treebanks such as English and Mandarin Chinese,[17] we can generate a PCFG and compact the rules for the parser (Krotov et al., 1999).

Initial steps have been taken towards the implementation of the second goal. The other goals are scheduled for implementation and testing in 2017.

5 Conclusion

While much of the FLE environment is under development, many components can be considered ready and usable. This includes the specification of grammar formalisms for the LFG parser backbone and XLE formalism parsers, the integration

[17] See for example the Chinese Treebank Project (Xue et al., 2002).

of various types of tokenizers and morphological analyzers, the CFG/PCFG backbones (and XLE rules) to a WFST-based grammar representation, two different types of parsing algorithms, and DAG-based unification.

Some of the currently implemented algorithms and components might be useful to other projects: a.) The LBNF specifications of the XLE grammar formalism. With BNFC these formal specifications can be used to generate parsers for the grammar formalisms in various other programming languages, including Java, Haskell, or C#. b.) The mapping of treebanks to PCFGs has been exported to a standalone tool. c.) A morphology-based language independent multi-word analyzer has been isolated as a standalone program.

We have performed preliminary performance tests using various Foma morphologies and the first parser implementation without unification and feature logic. Currently the English morphology as specified in a previous section, can process more than 150,000 tokens per second on a personal computer with an Intel Core i7 CPU and a current Linux distribution using GCC/G++ 6.x. This performance includes only covered vocabulary with lexical ambiguities and no guesser.

The WFST backbone based syntactic parser was tested on a small grammar with structural and lexical ambiguities and preceding tokenization and morphological analysis of all tokens. It parses approx. 3,000 sentences per second with an average sentence length of 7 words using the same architecture as described above. Unification is not included in the performance tests yet. This suggests that an improved version of such a parser can be expected to perform even faster in a final release.

The code of FLE is split into two sections. We have released public code in the Bitbucket repository at: https://bitbucket.org/dcavar/fle/. The development repository is open to team members by invitation only. If you want to join the development team, please send us an email and we will share the Bitbucket repository with you.

References

Allauzen, Cyril, Michael Riley, Johan Schalkwyk, Wojciech Skut & Mehryar Mohri. 2007. 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)*, vol. 4783 Lecture Notes in Computer Science, 11–23. Prague, Czech Republic: Springer.

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

Beesley, Kenneth R. & Lauri Karttunen. 2003. *Finite state morphology*. CSLI Publications. http://www.fsmbook.com.

Berstel, Jean & Christophe Reutenauer. 1988. *Rational series and their languages*. Berlin, New York: Springer-Verlag.

Bird, Steven, Ewan Klein & Edward Loper. 2009. *Natural language processing with python: Analyzing text with the natural language toolkit*. O'Reilly Media.

Bresnan, Joan. 2001. *Lexical-functional syntax*. Blackwell.

Butt, Miriam, Tracy Holloway King, María-Eugenia Niño & Frédérique Segond. 1999. *A grammar writer's cookbook*. CSLI Publications.

Cavar, Damir, Ivo-Pavao Jazbec & Siniša Runjaić. 2008. Interoperability and rapid bootstrapping of morphological parsing and annotation automata. *Informatica* 33(1). 107–113.

Clément, Lionel. 2016. XLFG Documentation. Technical report LaBRI. https://hal.archives-ouvertes.fr/hal-01277648.

Clément, Lionel & Alexandra Kinyon. 2001. XLFG-an LFG parsing scheme for french. In Miriam Butt & Tracy Holloway King (eds.), *Proceeding of the LFG01 Conference*, CSLI Publications.

Corkill, Daniel D. 1991. Blackboard systems. *AI Expert* 6(9). 40–47.

Crouch, Dick, Mary Dalrymple, Ron Kaplan, Tracy King, John Maxwell & Paula Newman. 2011. XLE documentation. Online document at http://www2.parc.com/isl/groups/nltt/xle/doc/xle_toc.html.

Dalrymple, Mary. 2001. *Lexical functional grammar* (Syntax and Semantics 42). New York: Academic Press.

Earley, Jay. 1968. *An efficient context-free parsing algorithm*: Carnegie-Mellon University dissertation.

Earley, Jay. 1970. An efficient context-free parsing algorithm. *Commun. ACM* 13(2). 94–102. doi:10.1145/362007.362035.

Emerson, Thomas. 2005. The second international Chinese word segmentation bakeoff. In *Proceedings of the Fourth SIGHAN Workshop on Chinese Language Processing*, vol. 133, 123–133.

Erman, Lee D., Frederick Hayes-Roth, Victor R. Lesser & D. Raj Reddy. 1980. The Hearsay-II speech-understanding system: Integrating knowledge to resolve uncertainty. *Computing Surveys* 12(2). 213–253.

Ferrucci, D. & A. Lally. 2004. UIMA: An architectural approach to unstructured information processing in the corporate research environment. *Natural Language Engineering* 10(3-4). 327–348.

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

Forsberg, Markus & Aarne Ranta. 2005. The labelled BNF grammar formalism. *Department of Computing Science, Chalmers University of Technology and the University of Gothenburg* .

Gamma, Erich, Richard Helm, Ralph Johnson & John Vlissides. 1995. *Design patterns: Elements of reusable object-oriented software*. Reading, Massachusetts: Addison-Wesley Publishing Company.

Hayes-Roth, Barbara. 1985. A blackboard architecture for control. *Artificial Intelligence* 26(3). 251–321. doi:10.1016/0004-3702(85)90063-3.

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

Jackendoff, Ray. 1997. *The architecture of the language faculty*. Cambridge, MA: MIT Press.

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

Jain, Anil K., Lin Hong & Sharath Pankanti. 2012. Ucto: Unicode tokenizer version 0.5.3 reference guide. Tech. Rep. ILK 12-05 Induction of Linguistic Knowledge Research Group, Tilburg Centre for Cognition and Communication, Tilburg University Tilburg, The Netherlands. https://ilk.uvt.nl/ucto/ucto_manual.pdf.

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

Kaplan, Ronald M. & Joan Bresnan. 1982. Lexical-functional grammar: A formal system for grammatical representation. Cognitive Theory and Mental Representation, 173–281. Cambridge, MA: The MIT Press.

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

King, Davis E. 2009. Dlib-ml: A machine learning toolkit. *Journal of Machine Learning Research* 10. 1755–1758.

Krotov, Alexander, Mark Hepple, Robert Gaizauskas & Yorick Wilks. 1999. Evaluating two methods for treebank grammar compaction. *Natural Language Engineering* 5(4). 377–394.

Kuich, Werner & Arto Salomaa. 1986. *Semirings, automata, languages* (EATCS Monographs on Theoretical Computer Science 5). Berlin, Germany: Springer-Verlag.

LeRoy Benjamin Sharon. 2016. Myanmar-Karen word lists. https://github.com/kanyawtech/myanmar-karen-word-lists.

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

Loper, Edward & Steven Bird. 2002. NLTK: The natural language toolkit. In *Proceedings of the ACL-02 Workshop on Effective Tools and Methodologies for Teaching Natural Language Processing and Computational Linguistics*, vol. 1, 63–70. Association for Computational Linguistics.

Marcus, Mitchell P., Mary Ann Marcinkiewicz & Beatrice Santorini. 1993. Building a large annotated corpus of english: The Penn treebank. *Comput. Linguist.* 19(2). 313–330. http://dl.acm.org/citation.cfm?id=972470.972475.

Maxwell, John & Ronald M. Kaplan. 1996a. An efficient parser for LFG. In *Proceedings of the First LFG Conference*, CSLI Publications.

Maxwell, John T. & Ronald M. Kaplan. 1991. A method for disjunctive constraint satisfaction. In Masaru Tomita (ed.), *Current Issues in Parsing Technology*, vol. 126, 173–190. Kluwer Academic Publishers.

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

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

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

Packard, Jerome L. 2000. *The morphology of Chinese: A linguistic and cognitive approach*. Cambridge University Press.

Poowarawan, Yuen. 1986. Dictionary-based Thai syllable separation. In *Proceedings of the Ninth Electronics Engineering Conference*, 409–418.

Roark, Brian, Richard Sproat, Cyril Allauzen, Michael Riley, Jeffrey Sorensen & Terry Tai. 2012. The OpenGrm open-source finite-state grammar software libraries. In *Proceedings of the ACL 2012 System Demonstrations*, 61–66.

Salomaa, Arto & Matti Soittola. 1978. *Automata-theoretic aspects of formal power series*. New York: Springer-Verlag.

Schling, Boris. 2011. *The Boost C++ libraries*. XML Press.

Shieber, Stuart M. 1985. Using restriction to extend parsing algorithms for complex-feature-based formalisms. In *Proceedings of the 23rd Annual Meeting on Association for Computational Linguistics* ACL '85, 145–152. Stroudsburg, PA, USA: Association for Computational Linguistics. doi:10.3115/981210.981228.

Tai, Terry, Wojciech Skut & Richard Sproat. 2011. Thrax: An open source grammar compiler built on OpenFst. Paper presented at the ASRU 2011.

The GATE Team. 2011. *Text processing with GATE (version 6)*. University of Sheffield Department of Computer Science.

Tseng, Huihsin, Pichuan Chang, Galen Andrew, Daniel Jurafsky & Christopher Manning. 2005. A conditional random field word segmenter for SIGHAN bakeoff 2005. In *Proceedings of the Fourth SIGHAN Workshop on Chinese Language Processing*, 168–171.

Wallach, Hanna M. 2004. Conditional random fields: An introduction. Tech. Rep. MS-CIS-04-21 University of Pennsylvania.

Xue, Nianwen, Fu-Dong Chiou & Martha Palmer. 2002. Building a large-scale annotated Chinese corpus. In *Proceedings of the 19th International Conference on Computational Linguistics (COLING 2002)*, Taipei, Taiwan.

Zhao, Hai, Chang-Ning Huang, Mu Li & Bao-Liang Lu. 2010. A unified character-based tagging framework for Chinese word segmentation. *ACM Transactions on Asian Language Information Processing (TALIP)* 9(2). 5.