FLE Preliminary Results

Damir Cavar, Lwin Moe, Hai Hu
Indiana University

Headlex 2016, Warsaw, Poland

Help

Graduate Students

- Hai Hu, Kenneth Steimel, Tim Gilmanov, Joshua Herring

Support

- Kenneth Beesley
- Lionel Clement
- Thomas Hanneforth
- Ronald Kaplan
- Gerald Penn
- Richard Sproat
- Annie Zaenen
- ...

Support

Provided morphologies and grammars to test:

- Mary Dalrymple
- Helge Dyvik and Paul Meurer
- Agnieszka Patujek and Adam Przepiórkowski

Morally supported and brought up the idea of the Monotonicity Calculus integrated in an LFG and/or CCG type of parser: Larry Moss

Local IU community: Sandra Kübler, Markus Dickinson

The BNFC-team fixed several compiler issues for our code generation.

Motivation

- Need for a *modern* grammar engineering platform
- Platform independent (e.g. Linux, OSX, Windows, Chrome OS, Android, iOS)
- Parallelizable and distributed architecture
- Interoperable
    - Tied to common scripting and web languages like Python, JavaScript.
    - Import and export standards/exchange formats using XML, JSON, etc.
- Open License (e.g. **Apache License 2.0**, MIT License)

Motivation

Purpose

- Computational Language Documentation
- Research and Education
- Productive development of applications
- Platform for hybrid white- and black-box modeling:
    - Grammar engineering combined with machine learning algorithms for probabilistic models or (grammar) induction.

Infrastructure

- Two Bitbucket Git repositories:
  - Private repo for experimenting, tutorials, data, etc.
    - Access via email and contact (write me!)
  - Open repository
    - https://bitbucket.org/dcavar/fle/
    - Not much there yet

Infrastructure

- Coding in C++11 and newer using
  - GCC/G++, Clang/LLVM, Xcode, Cygwin, MS VisualStudio.
  - CMake-based compiler configuration.
- BNFC-based grammar to code conversion (using flex and bison).
- Doxygen-based code documentation.
- Git-based code and version management (using Bitbucket).
- CLion IDE.
- OS: Linux, Mac, Windows

Code and Dependencies

- Required libraries (so far):
  - C++ Standard Library
  - Boost Libraries
  - Foma
- In the final version also:
  - OpenFST
  - OpenGrm Thrax Grammar Development Tool

Code and Interoperability

- The following libraries will be optionally linked:
  - Ucto – Unicode rule-based tokenizer
  - Alternative FST-libraries (e.g. HFST)

- Required and optional libraries are available and/or made available on the main desktop operating systems (all are C or C++ based).

Goals

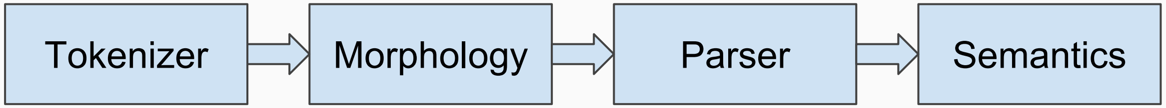
- Library of services rather than monolithic parser or toolset:
    - Parsing CFG, PCFG, CCG and related formalisms
    - Parsing XLE compatible grammars
    - Utilizing XFST-compatible morphologies (using e.g. Foma)
        - Conversion of XFST-morphology outputs to various formats
    - Tokenizers using Foma-based FSTs, rule-based tokenizers for Ucto, simple regular expression based tokenizers
    - Parsing-algorithms that use the different formalisms above

Goals

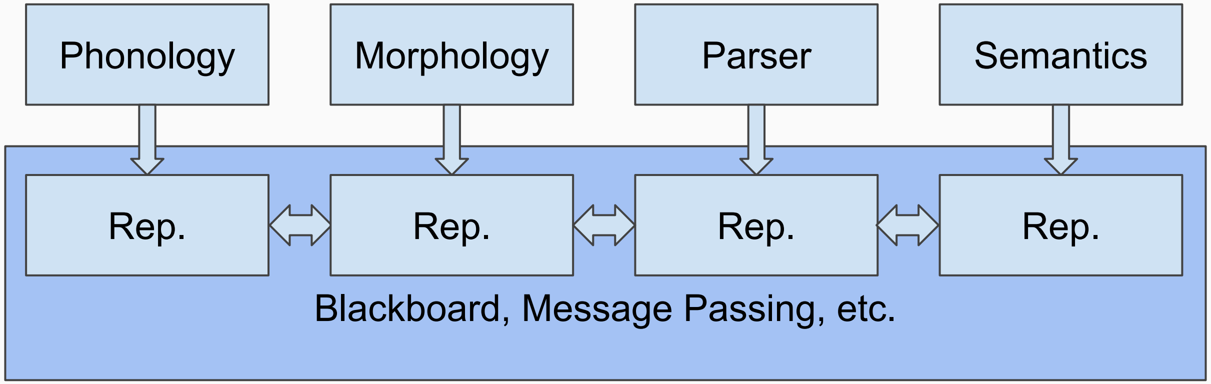
- Library of services:
  - Relating to Dependency Grammars (mapping from c- and f-structures)
  - Integration of training and machine learning algorithms: probabilistic grammar backbone, morphologies, c- and f-structure relations
  - Available for C++-code base and as modules to common scripting languages

Application

Classical pipeline architecture:



Parallel architecture with mapping constraints (Jackendoff, 1997, 2007):



Current implementation: Tokenization

- Simple space-based (regular expressions, Boost)
- Foma-based (e.g. for Burmese and related languages)
- Ucto-based possible, not tested yet

Current implementation: Morphology

- Foma-based (e.g. for English, Croatian, Burmese, Mandarin)
  - Processing of approx. 200,000 ambiguous tokens per second within the parser integration (using 3rd gen. Intel i7 laptop CPU on a single thread/core)
- Potentially also:
  - Interface to simpler Part-of-Speech taggers.

Current implementation: Syntactic Parsing

- Simple Earley-type of Parser using hash-tables for rules and edges
    - *Prediction, Scanning, Completion*
    - Edges as indexed dotted rules on a chart/stack
    - Unification over trees with root or goal symbol
- Weighted Finite State Transducer (WFST) as grammar representation

Toy Rules

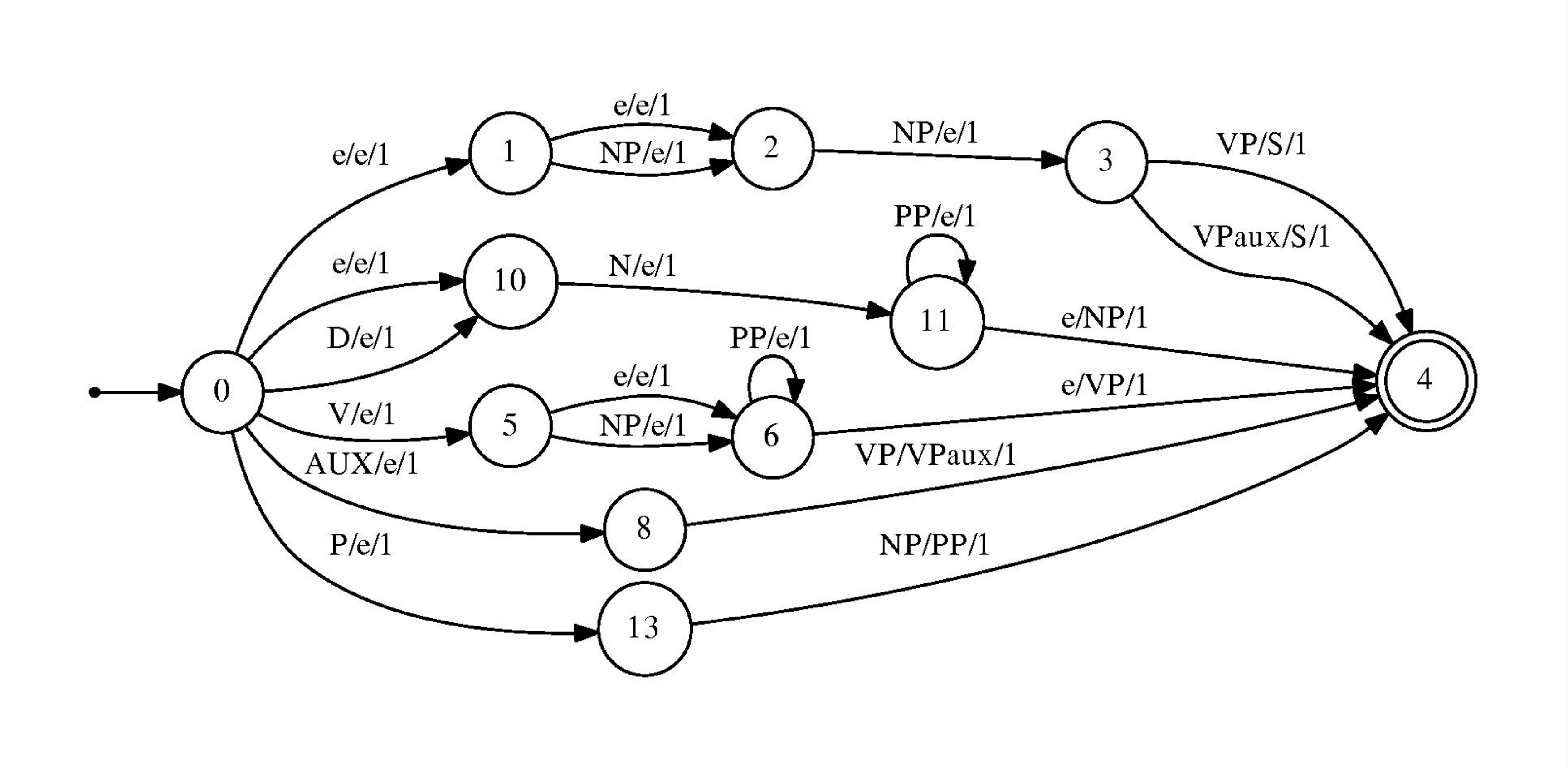
```
TOY  ENGLISH  RULES (1.0)

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

   VP --> V
        (NP: (^ OBJ)=!
             (! CASE)=ACC)
        PP*:! $ (^ ADJUNCT).

   VPaux --> AUX
             VP.

   NP --> (D)
        N
        PP*:! $ (^ ADJUNCT).

   PP --> P
        NP:(^ OBJ)=!
           (! CASE)=ACC.
```

Grammar Backbone as a WFST

T as a 7-tuple $(Q, \Sigma, \Gamma, I, F, \lambda, \varrho)$ with

- Q a finite set of states
- Σ a finite set over the input alphabet
- Γ a finite set over the output alphabet
- I a subset of Q of initial states (only one in our case)
- F a subset of Q of final states
- $E \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times (\Gamma \cup \{\varepsilon\}) \times Q \times K$, a mapping of a state $\in Q$ and an input symbol $\in \Sigma \cup \{\varepsilon\}$ to an output symbol $\in \Gamma \cup \{\varepsilon\}$ and a new state $\in Q$; and $\lambda : I \rightarrow K$ mapping initial states and $\varrho : F \rightarrow K$ final states to weights.

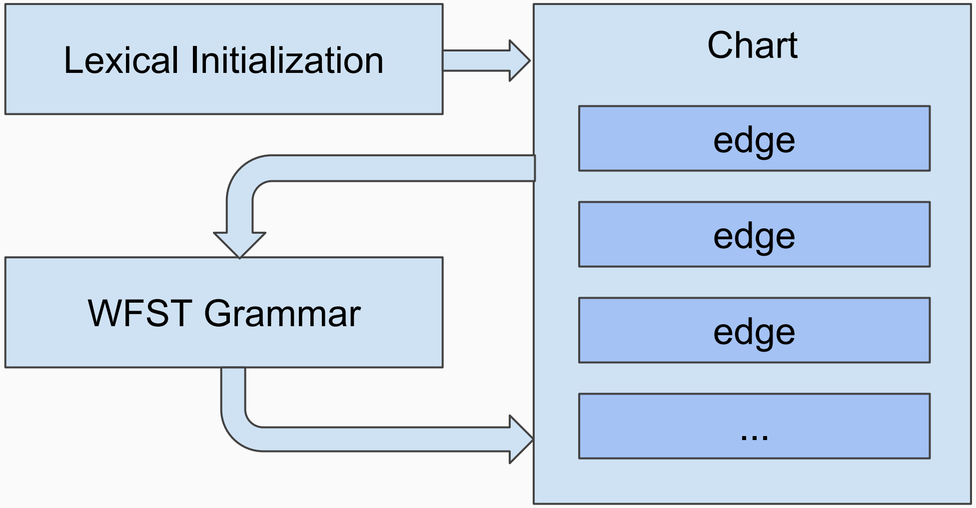
Grammar Backbone as a WFST



Cavar et al. (2016): Free Linguistic Environment Preliminary Results

WSFT Backbone

Similar to Earley algorithm:



Cavar et al. (2016): Free Linguistic Environment Preliminary Results

WSFT Backbone

Implementation:

- Edges are integer tuples, i.e. indexes over input token vectors and states in the WFST.
- WFST own class with simple optimization.
- Slower than simple Earley-type of implementation.

Weights:

- Probabilities of rules as in PCFGs.
- Transitions of symbols as in Markov Chains
- Unification and AVMs
- A combination of all the above

WFST Extensions

- Export of DOT specification (and indirectly SVG, PDF, etc.).
- Binary dump of WFST for faster load cycles.

- Reimplementation of WFST based on OpenFST with the benefits of the rich set of library functions.
- Extension with OpenGrm, i.e. an OpenFST-based implementation of a single- and double-stack pushdown automaton.

Restricted Backbone as WFST

Potentially:

- Limited recursion depth for center embeddings, and
- Mapping of CFG backbone to a WFST with all possible word order regularities.
- Generation of a very efficient parser with certain limitations of the backbone complexity.

WFST Backbone and Parser

Current grammar formalisms defined in LBNF and converted with BNFC to C++ parsers:

- CFG
- PCFG
- XLE
    - CONFIG (complete)
    - FEATURES (incomplete)
    - LEXICON (incomplete)
    - MORPHOLOGY (incomplete)
    - TEMPLATES (missing)
    - RULES (no: edit rules, METARULEMACRO, …)

LBNF and Formalisms

```
comment "\"" "\"" ;
Grammar.    GRAMMAR ::= [RULE] ;

RuleS.                RULE ::= WORD [LEXDEF] ;
RuleSDisjunction. RULE ::= WORD "{" [DLEXDEF] "}" ;
RuleUnknown.          RULE ::= "-unknown" [LEXDEF] ;
RuleToken.            RULE ::= "-token" [LEXDEF] ;
RuleSEditEntry.           RULE ::= WORD [EDITENTRY] ;
RuleUnknownEditEntry. RULE ::= "-unknown" [EDITENTRY] ;
RuleTokenEditEntry.   RULE ::= "-token" [EDITENTRY] ;
terminator RULE "." ;
Definition.           LEXDEF ::= CAT MORPHCODE [DSCHEMA] ;
DefinitionSimple. LEXDEF ::= Label ;
separator             LEXDEF ";" ;
DefinitionDisjunct. DLEXDEF ::= LEXDEF ;
separator DLEXDEF "|" ;
...
```

BNFC Output

```
void Skeleton::visitGrammar(Grammar *grammar) {
  /* Code For Grammar Goes Here */
  grammar->listrule_->accept(this);
}

void Skeleton::visitRuleS(RuleS *rules) {
  /* Code For RuleS Goes Here */
  rules->word_->accept(this);
  rules->listlexdef_->accept(this);
}

void Skeleton::visitRuleSDisjunction(RuleSDisjunction *rulesdisjunction) {
  /* Code For RuleSDisjunction Goes Here */
  rulesdisjunction->word_->accept(this);
  rulesdisjunction->listdlexdef_->accept(this);
}
```

LBNF and Formalisms

BNFC

- Haskell-based BNF Converter to flex and bison code.
- Compilation using C++ compiler (if conversion to C++).
- Generates LaTeX documentation of parser definition.
- Generates test-binaries for testing formalism/language parser.
- Generates a parser class using the visitor-architecture.

Current implementation: Unification

- Basic algorithm using Directed Acyclic Graphs (DAG)
- No advanced algorithms yet, e.g. Disjunction, Constraints, Negation
- No performance tests
- Considerations:
  - Optimization using mapping of AVMs to bit-vectors for unification
  - Caching of operations and results
  - Unification over resulting c-structures or during transitions using WFSTs

TODOs

- **Windows**
  - So far using Cygwin, preparing to use native DLLs:
    - We need a setup to generate Boost, Foma, OpenFST, OpenGrm as DLLs
    - Adaptation of the CMake code
- **Mac OS X**
  - Similar library-requirements as Windows, but much easier to compile native linking libraries (using Clang and the LLVM compiler environment that comes with XCode)

TODOs

- **Compiling libraries**
  - Separation of an application specification and environment from core functionalities that could be defined in libraries only.
  - Definition of a Python 3.x extension module, i.e. the Grammar engineering environment could be written in Python and Qt or JavaScript and NodeJS for example.

TODOs

XLE-formalism

- Finalize all grammar section parsers with coverage for all sample grammars that we have.

Parser algorithm

- Finalize the two different parsing with unification during edge formation or after parse tree generation for complete parse trees only and evaluate behavior and performance.

And a lot more...

Related activities as part of FLE

Morphologies:

- English
- Croatian (port of old CroMo with Ragel-based rule compiler)
- Burmese (and related languages)
- Mandarin
- and integration of other freely available morphologies