PyFoma: a Python finite-state compiler module

Mans Hulden[3] **Michael Ginn**[3] **Miikka Silfverberg**[ʕ] **Michael Hammond**[ʁ]
[3]University of Colorado [ʕ]University of British Columbia [ʁ]University of Arizona
first.last@colorado.edu first.last@ubc.ca hammond@arizona.edu

Abstract

We describe PyFoma, an open-source Python module for constructing weighted and unweighted finite-state transducers and automata from regular expressions, string rewriting rules, right-linear grammars, or low-level state/transition manipulation. A large variety of standard algorithms for working with finite-state machines is included, with a particular focus on the needs of linguistic and NLP applications. The data structures and code in the module are designed for legibility to allow for potential use in teaching the theory and algorithms associated with finite-state machines.

1 Introduction

Finite-state technology is well established in the NLP community and is widely used for various lower-level tasks such as tokenization, morphology, phonology, spell checking, spelling correction, grapheme-to-phoneme (G2P) mapping, various speech applications, and general string processing (Roche and Schabes, 1997; Mohri et al., 2008; Hulden, 2022). In tandem with neural models, finite-state models can also be used to constrain the output of a neural network to prevent generation of text that fails to adhere to a specific format (Ghazvininejad et al., 2017). Furthermore, finite-state methods may be effective in low-resource scenarios, allowing the incorporation of expert knowledge (Moeller et al., 2019; Muradoglu et al., 2020; Beemer et al., 2020).

Several tools exist for constructing and manipulating finite-state automata (FSAs) and transducers (FSTs), together finite-state machines (FSMs).[1] Some, such as *OpenFST* (Allauzen et al., 2007), *Carmel* (Knight and Graehl, 1998), *foma* (Hulden, 2009b), and *xfst* (Beesley and Karttunen, 2003) are stand-alone tools, implemented in C or C++ for efficiency reasons, while others, such as *Pynini* (Gorman, 2016), *HFST* (Lindén et al., 2009), and *Kleene* (Beesley, 2012) rely on the existing algorithms and APIs of *OpenFST* or *foma* to allow for FST manipulation in higher-level languages.

[1]As every FSA can be expressed as an FST and vice-versa, we often use these terms roughly interchangeably.

```
from pyfoma import FST
fst = FST.re("d a:(æ<1>|(eɪ)<4.5>) (ta):(ɾə)")
fst.view()
```

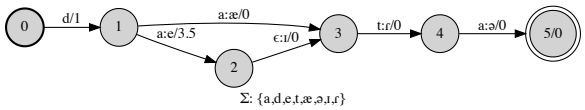


Figure 1: A weighted finite-state transducer compiled and visualized in PyFoma.

PyFoma is a complete finite-state toolkit written in pure Python. It is available on GitHub at https://github.com/mhulden/pyfoma, and easily installable via the PyPi package repository at https://pypi.org/project/pyfoma/. The project has the following overall aims:

- Simple and intuitive to use and integrates with the interactive mode of development of Jupyter notebooks
- Provides a comprehensive suite of algorithms for constructing and manipulating weighted and unweighted FSMs
- Incorporates visualization capabilities to examine FSMs as they are constructed
- Includes an extensible regular expression-to-FSM compiler with a formalism that is a close as possible to standard formalisms, such as that of the Python `re`-module
- Provides implementations of algorithms that are similar to pseudocode and can be used in instructional settings to understand FSM algorithms in detail.

Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations), pages 258–265

```
var1 = FST.re("cat")
var2 = FST.re("c a t")                          # same as above
var3 = FST.re("\+ \* \ ")                       # literalizing special characters
var4 = FST.re("'+' '*' ' '")                    # same as above using single quotes
var5 = FST.re("(cat|dog|mouse)s?")
var6 = FST.re("[A-Za-z0-9] - [aeiouAEIOU]")     # all ASCII characters, except vowels
var7 = FST.re("[^aeiou]")                       # any symbol, except lowercase vowel
var8 = FST.re("(cat):(gato) @ (gato):(chat)") # cross-product (:), composition (@)
var9 = FST.re("cat<1.0>|dog<2.0>|mouse<3.0>") # weights specified by <float>
```

Table 1: PyFoma basic regular expression examples for compilation of automata and transducers.

- Is efficient enough so that large-scale morphological analyzers can be compiled into FSMs with reasonable speed so that the dependency to low-level command-line tools is eliminated
- Incorporates advanced string rewriting rule compilation to allow for compilation of phonological and morphophonological rules in the vein of Chomsky and Halle (1968)
- Includes a large assortment of practical examples in the documentation for a variety of linguistic tasks
- Allows extension of the regular expression formalism where users can define new regular expression operators and implement them in Python

A demo video is available on YouTube.[2]

2 Illustrative example

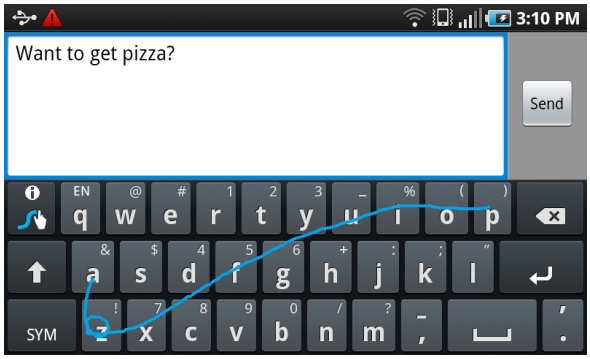


Figure 2: A Swype swiping action—to decode a Swype one must calculate what English word could be hiding inside the sequence `poiuygfdcxza`.

A typical low-level NLP task which FSTs are particularly well suited for is *Swype* decoding. A user swipes a finger across a virtual keyboard, touching a sequence of letters along the way, such as `poiuygfdcxza` (see Figure 2). The decoding task is to retrieve all valid English words that could have been intended by the user. Such a *Swype*-decoder can be constructed through the composition of three transducers, in four short lines of PyFoma code (see Figure 3). To achieve such succinct solutions, it is crucial to have a regular expression compiler that implements a variety of algorithms to manipulate weighted and unweighted automata and transducers.

3 Implementation

The core functionality—apart from visualization—of PyFoma is implemented entirely in Python,[3] relying only on the Python standard library for its dependencies, and exposing a Pythonic API. PyFoma provides a class `FST` with methods to construct (W)FSAs/(W)FSTs out of either regular expressions or algebraic manipulation.

3.1 Regular expression parser and compiler

The simplest PyFoma regular expression is a string like `cat`.[4] These simple expressions can be combined using regular expression operators. In contrast to many other finite-state tools, PyFoma uses standard operators found in pattern matching applications such as Python's `re` module, allowing for succinct FST construction (Table 1). These include union (`|`), Kleene star (`*`), Kleene plus (`+`), optionality (`?`), and character classes, such as `[A-Za-z]`. To aid in legibility of complex regular expressions, whitespace is not significant and any actual spaces must be escaped by `\␣` or `'␣'`.

[2] https://www.youtube.com/watch?v=X4ovo7phrV0

[3] While it shares part of the name of the *foma*-tool (written in C, includes Python bindings), PyFoma inherits none of the code in foma, but does use the main ideas for constructing rewrite rule transducers in Hulden (2009a).

[4] By default, the regex compiler treats each character as an individual token. This behavior can be overridden by surrounding strings in single quotes; the compiler will then treat the entire string as a token.

```
In [1]: from pyfoma import FST
        import string

        fsts = {}
        fsts['w'] = FST.from_strings(open("engwords.txt")) # wordlist
        fsts['removedouble'] = FST.re("(" + '|'.join(f"{l}+:{l}" for l in string.ascii_lowercase) + ")*")
        fsts['insert'] = FST.re(". ('':. | .)* .") # repeat one, then insert or repeat, repeat last
        fsts['swype'] = FST.re("$w @ $removedouble @ $insert", fsts)

        max(list(fsts['swype'].analyze("poiuygfdcxza")), key = len)

Out[1]: 'pizza'
```

Figure 3: Solving a key part of a Swype keyboard application—calculating what set of English words would be compatible with a fingerswipe over the letters *poiuygfdcxza*—can be accomplished by composition of FSTs that (a) repeat English words (e.g. `pizza`) and (b) remove doubled letters (e.g. `pizza` → `piza`) and arbitrarily insert letters in between the first and last letters (e.g. `piza` → `poiuygfdcxza`). The composition of these transducers constructs a Swype decoding transducer that directly maps `poiuygfdcxza` to the set of valid English words `pizza` and `pa`.

PyFoma implements several additional operators not found in pattern matching regexes: intersection (`&`), set subtraction (`-`), cross-product (`:`), optional cross-product (`:?`), relation composition (`@`), and weight specification with angled brackets (e.g. `<1.0>`).

Variables A typical workflow for constructing FSMs for NLP applications is to build a final FSM piece-by-piece, storing the intermediate steps as variables, which are combined with regex operations to build more complex FSMs. To achieve this, the `FST.re` can be passed a `dict` argument that instructs the compiler how to find the variables, which themselves are prefixed by the sigil `$`. The following is a typical sequence:

```
fsts = {} # init dict to store variables
fsts['V'] = FST.re("[aeiou]")
fsts['C'] = FST.re("[a-z]-$V", fsts)
fsts['syll'] = FST.re("$C+ $V $C+", fsts)
```

Built-in Functions The compiler provides some functionality through built-in functions (Table 2) rather than regular expression operators. Similarly to variables, functions are distinguished by the `$^`-sigil. For example, reversal of an FSM is invoked as `$^reverse(fsm)`.

Customizing the compiler The regular expression compiler can be further customized by the user by defining additional functions written in Python that the compiler will call when compiling. For example, suppose we needed a function that made all states in an FSM final with weight 0.0. We can define a function in Python that does so as follows:

```
def allfinal(myfsm):
    for s in myfsm.states:
        s.finalweight = 0.0
    myfsm.finalstates = myfsm.states
    return myfsm
```

The compiler can then be invoked with the keyword argument `functions`, which is a set of the user-specified functions that the compiler should be aware of, for example:

```
custom = {allfinal}
FST.re("$^allfinal(cat|dog)",
       functions = custom)
```

Naturally, the custom function could itself call the regex compiler—a common way of defining customized behavior. For example, an often used idiom in regexes is the pattern `.* X .*`, i.e. the set of strings that **contains** `X` as a substring, where `X` is an arbitrary FSM. We could create a new function `$^contains()` as follows.

```
def contains(fst):
    return FST.re(".* $X .*", {'X':fst})
```

Compiler behavior To simplify usage, the compiler always returns FSMs that are **determinized**, **minimized**, and **coaccessible**. Determinization and minimization is performed periodically during compilation of intermediate FSMs as well. In the weighted case, weights are pushed as close as possible to the initial state by a **weight pushing** algorithm (the effect of this is seen in Figure 1). Since transducers (as opposed to automata) are not guaranteed to be determinizable, transducer determinization treats a transition tuple with several dimensions as a single symbol. Weighted automata, likewise, are not guaranteed to be determinizable, and are therefore pseudo-determinized so that the weight becomes part of the label. True WFSA determinization is available through the API function `determinized()`, which, however, will not terminate for undeterminizable WFSAs.[5]

[5] An algorithm exists for testing determinizability (Allauzen and Mohri, 2003); however, employing it results in much slower compilation times overall.

```
$^determinize(fsm)                          # determinizes an FSM
$^ignore(x,y)$                              # The language x, ignoring intervening y's
$^project(fsm, dim)                         # extract a projection
$^invert(fsm)                               # inverts a transducer
$^minimize(fsm)                             # minimizes an FSM
$^not(fsa)                                  # complement of FSA
$^input(fsm) or $^output(fsm)               # extract intput or output projection
$^project(fsm, dim)                         # project one of the tapes in FST
$^restrict(a / b _ c,...)                   # context restriction compilation
$^reverse(fsm)                              # reverses an FSM
$^rewrite(a:b)                              # basic rewrite rule compilation
$^rewrite(a:b / c _ d, e _ f, ...)          # basic rewrite rule with contexts
$^rewrite(a:?b / c _ d)                     # optional rewrite rule
$^rewrite(':x a ':y / c _ d)                # 'markup' rule (wrap x, y around a)
$^rewrite(a:b / c _ d, leftmost = True)     # always use leftmost possible rewrite
$^rewrite(a:b / c _ d, longest = True)      # always use longest possible rewrite
$^rewrite(a:b / c _ d, shortest = True)     # always use shortest possible rewrite
```

Table 2: Some PyFoma regular expression compiler operations are expressed as functions instead of an operator, such as `|` or `*`. Rewrite rule specifications allow multiple modalities and fine-grained control over rewriting transducers.

3.2 Transducer operations/algorithms

The PyFoma module provides 26 operations on FSMs as well as additional operations that provide information about FSMs and their paths, languages and relations. All of the algorithms are available in *mutating* (methods that modify the original FSM) and *non-destructive* versions, following Python naming conventions.[6] These include concatenation, union, intersection, subtractions, composition, cross-product, Kleene closures, reversal, inversion, transducer projections, weighted and unweighted determinization, minimization, epsilon-removal, weight pushing, n-best path extraction, inter alia. While these are usually called through associated methods (e.g. `myfst.reverse()`), many of the fundamental operations are also available through overloaded Python operators: e.g. `fsm1 | fsm2` denotes the union of two FSMs, `fsm1 @ fsm2` the composition, etc. Additionally, there are operations for building FSMs from lists of strings.

The weight calculus is by default performed in the widely-used *tropical semiring* (Pin, 1998) where the weights along a path are summed, and weights across parallel paths with the same labeling are subject to the `min()`-operation, similar to the Viterbi assumption in probabilistic models. Also supported is the log semiring, which replaces the `min()`-operation with logspace addition, making the weight behavior similar to working with negative log-probabilities. However, operations are often much slower with the log semiring, explaining the popularity of the tropical semiring.

Transducers are not internally constrained to be 2-tape transducers. Since the labels on FSM transitions are arbitrary Python tuples, PyFoma can represent multi-tape automata as well. These have been shown to be useful in modeling, for example, intermediate steps in a sequence of historical sound changes (Hulden, 2017).

We have strived to maintain data structures and object naming that facilitate writing pseudocode-like implementations of the algorithms. Apart from minor bookkeeping, the implementations are often similar in length as the pseudocode in sources such as Mohri (2009) that describe WFSA/WFST algorithms. Figure 4 illustrates a method in PyFoma.

```
def label_states_topology(self, mode = 'BFS'):
    """Topologically sort, label states."""
    cntr = itertools.count()
    Q = deque([self.initialstate])
    inqueue = {self.initialstate}

    while Q:
        s = Q.pop() if mode == 'DFS'\
                    else Q.popleft()
        s.name = str(next(cntr))
        for label, t in s.all_transitions():
            if t.targetstate not in inqueue:
                Q.append(t.targetstate)
                inqueue.add(t.targetstate)
    return self
```

Figure 4: Example algorithm from the PyFoma codebase—topological sorting of states in an FSM, either by breadth-first-search (BFS) or depth-first-search (DFS).

[6] For example, the mutating function is `fst.reverse()` while the non-destructive version is `reversed(fst)`.

3.3 Rewrite rules

Rewrite rules—the formalism popularized by *The Sound Pattern of English* (SPE) (Chomsky and Halle, 1968), are commonly used for describing phonological and morphophonological alternations, and PyFoma includes a variety of rule-to-FST algorithms. The ability to compile such rules to FSTs is usually a core requirement to be able to construct linguistically sophisticated rule-based morphological analyzers and generators.

The simple use case is generally of the type `$^rewrite(a:b, c _ d)`, which describes the rule "rewrite instances of `a` to `b` when occurring between `c` and `d`," expressed in the phonological literature as $a \rightarrow b / c \_ d$. Multiple comma-separated contexts are also possible. The left-hand side can be an arbitrary transducer, although it is usually constructed with the cross-product (`:`). While this covers many needs, some applications, such as chunking and markup (Beesley and Karttunen, 2003) as well as syllabification (Hulden, 2005), require more fine-grained guidance to control the rewriting. For example, one may want to rewrite *as little as possible* or, conversely, *as much as possible* if there is ambiguity in what the left-hand-side of the rule (`a`) denotes. Table 2 gives a brief overview of the main modalities available; these cover various types of string rewriting suggested in the literature (Kaplan and Kay, 1994; Beesley and Karttunen, 2003; Hulden, 2009a). Weights can also be integrated into the rules, modeling rules that have a cost associated with applying them, e.g. `$^rewrite(b:p<2.0> / _ #)`, which describes the rule "devoice a `b` at the end of a word with cost 2.0".

3.4 Right-linear grammars

Many linguists favor modeling the lexicon component of morphological analyzers as a right-linear grammar which captures the morphotactics of a language. This lexicon component is then normally composed with a battery of rewrite rules that handle morphophonological alternations, producing the full grammar transducer. Earlier tools, such as `lexc` (Karttunen, 1993) and `lexd` (Swanson and Howell, 2021) are strongly influenced by the formalism of right-linear grammars. PyFoma includes a right-linear grammar compiler, very similar to *lexc*, which is also found in the earlier tools *xfst* and *foma*.

3.5 Feature calculus

Construction	Description
`[[$X=y]]`	Set `X` to value `y`
`[[$X=]]`	Unset (or clear) `X`
`[[$X?=y]]`	Unify `X` with value `y`
`[[$X==y]]`	Check that `X` equals `y`
`[[$X!=y]]`	Check that `X` does not equal `y`
`[[$X]]`	Check that `X` has been set
`[[!$X]]`	Check that `X` has not been set

Table 3: PyFoma feature calculus. Here `X` is a variable and `y` a value.

FSMs usually suffer from the difficulty of modeling long-distance dependencies between symbols. For example, for a language that includes circumfixes in its morphology, the section of the FSM that models the word stems generally doubles in size, since states must be duplicated along circumfixing and non-circumfixing paths.

PyFoma supports a type of *feature calculus* where specific feature-value queries and checks can be used to control long-distance matches or mismatches. Figure 5 shows a minimal example of two automata that accept two German verb forms of **hören** 'to hear', the citation form and the past participle **gehört**. The first automaton needs to double the complete path of the stem **hör**, while the second, which contains feature-value setting and checking can share the stem parts. In the second example, a feature (arbitrarily) called `pp` is set to value `1` (`[[$pp=1]]`) for the prefix part of the circumfix, and is later required (`[[$pp==1]]`) to have the value one for the matching suffix path, and disallowed (`[[$pp!=1]]`) to have that value for the non-circumfix path. Such feature-value symbol manipulation can aid in the construction of grammars for languages with many morphological long-distance dependencies. They can also be removed from an FSM, and an equivalent FSM—possibly larger—is calculated. See Table 3 for the full inventory of supported feature constructions.

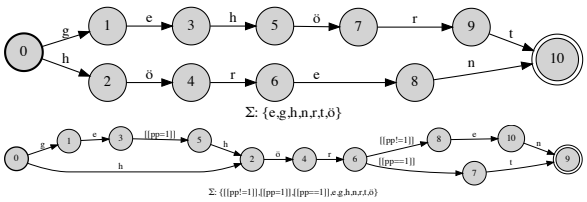
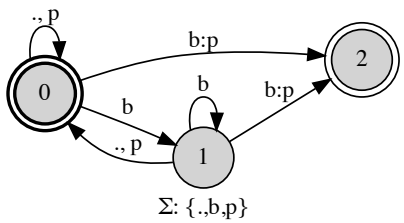


Figure 5: Illustration of how feature calculus strategies can model long-distance dependencies.

Our feature-notation is inspired by attribute-value feature unification schemes in syntactic theory (Sag et al., 1986) and the implementation of "flag diacritics" in the *xfst*-tool (Beesley and Karttunen, 2003).

3.6 Visualization

```
re("$^rewrite(b:p /  _ #)").view()
```




```
(re("kæb") @ re("$^rewrite(b:p /  _ #)")).view()
```

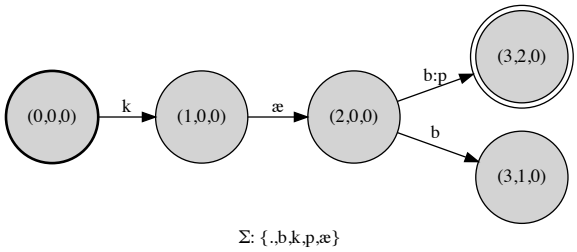


Figure 6: An example of the composition algorithm illustrating through state-name triplets in the result how the composition algorithm operates.

PyFoma depends on Graphviz (Ellson et al., 2002) for visualization of FSMs. `FST` objects have an associated `view()` method, which allows for visualization in a notebook, and a `render()` method, which generates a PDF. Graphviz is installed automatically with PyFoma if PyPi is used.

When called outside the regex compiler, many of the fundamental algorithms also provide illustrative state naming to show how the algorithms operate when combining two FSMs, which may be useful for teaching purposes. For example, Figure 6 shows in the state names, which are triplets of numbers, how two FSMs are combined through composition by traversing both in parallel and keeping track of matching input and output transitions in the two FSMs' states. In the example, the first number represents the state of the FSM created from the string kæb ($0 \xrightarrow{k} 1 \xrightarrow{æ} 2 \xrightarrow{b} 3$), the second represents the state of the transducer `$^rewrite(b:p / _ #)` (devoice b at the end of a string), and the third represent the state of a filter transducer with three states that is used internally during composition to eliminate redundant paths (Mohri, 2009). A similar visualization is done when transducers are compiled from right-linear grammars. In that case, the names of the states are the left-hand side of the right-linear grammar rule, e.g. `Noun` for a rule such as `Noun` → `cat|dog|bus`.

3.7 Context-free grammars

It is often both possible and desirable to approximate *context-free grammars* with finite-state models (Evans, 1997; Mohri and Nederhof, 2001). PyFoma includes preliminary support for parsing and visualizing context-free grammars in several formalisms.

4 A Morphological Analyzer Example

Figure 7 shows a mini-grammar built with the same principles as larger grammars. First, a lexicon component is constructed as a transducer (the concatenation `$noun $infl`). Following this, a sequence of rewrite-rule transducers are composed with the lexicon. In our case we have two rules: the first inserts `e` between a sibilant consonant on the left and `+s` on the right (e.g. `bus+s` → `buse+s`); the second removes all `+`-symbols which are used temporarily to denote morpheme boundaries. The composite transducer represents a minimal example of an analyzer/generator.

```
from pyfoma.fst import re

fsts = {}
fsts['noun']  = re("cat|dog|bus|fox")
fsts['infl']  = re("'[Sg]':'' | '[Pl]':(\+s)")
fsts['sib']   = re("[szx]|ch|sh") # Sibilants
fsts['rule']  = re("$^rewrite(':e / $sib _ \+ s)", fsts)
fsts['clean'] = re("$^rewrite(\+:'')")
grammar       = re("$noun $infl @ $rule @ $clean", fsts)

print(list(grammar.generate("fox[Pl]")))
print(list(grammar.analyze("buses")))
```

```
['foxes']
['bus[Pl]']
```

Figure 7: A toy analyzer/generator snippet that handles English noun inflection and e-insertion. The last two lines show the `generate` and `analyze` methods for FSTs.

5 Conclusion

We introduce PyFoma, an open-source Python module to facilitate the construction of weighted and unweighted FSMs, with specific support for NLP applications. We hope to provide a stable module that features a broad set of tools for use in teaching, research, and development of applications using finite-state technology and includes detailed documentation, example code, tutorials, and exercises.

Acknowledgements

We are deeply grateful to Ken Beesley for expert guidance regarding multiple aspects of design and syntax in NLP-oriented finite-state libraries. The PyFoma project was originally inspired by the *Kleene* library (Beesley, 2012) developed by Ken at SAP Labs and has adopted some of its regular expression syntax and ideas about integration of weights into regular expressions and rewrite rules. We have tried to follow Ken's motto: "Syntax Matters".

References

Cyril Allauzen and Mehryar Mohri. 2003. Efficient algorithms for testing the twins property. *Journal of Automata, Languages and Combinatorics*, 8(2):117–144.

Cyril Allauzen, Michael Riley, Johan Schalkwyk, Wojciech Skut, and Mehryar Mohri. 2007. OpenFST: A general and efficient weighted finite-state transducer library: (extended abstract of an invited talk). In *Implementation and Application of Automata: 12th International Conference, CIAA 2007, Praque, Czech Republic, July 16-18, 2007, Revised Selected Papers 12*, pages 11–23. Springer.

Sarah Beemer, Zak Boston, April Bukoski, Daniel Chen, Princess Dickens, Andrew Gerlach, Torin Hopkins, Parth Anand Jawale, Chris Koski, Akanksha Malhotra, Piyush Mishra, Saliha Muradoglu, Lan Sang, Tyler Short, Sagarika Shreevastava, Elizabeth Spaulding, Testumichi Umada, Beilei Xiang, Changbing Yang, and Mans Hulden. 2020. Linguist vs. machine: Rapid development of finite-state morphological grammars. In *Proceedings of the 17th SIGMORPHON Workshop on Computational Research in Phonetics, Phonology, and Morphology*, pages 162–170, Online. Association for Computational Linguistics.

Kenneth R. Beesley. 2012. Kleene, a free and open-source language for finite-state programming. In *Proceedings of the 10th International Workshop on Finite State Methods and Natural Language Processing*, pages 50–54, Donostia–San Sebastián. Association for Computational Linguistics.

Kenneth R. Beesley and Lauri Karttunen. 2003. *Finite-state morphology: Xerox tools and techniques*. CSLI, Stanford.

Noam Chomsky and Morris Halle. 1968. *The Sound Pattern of English*. Harper & Row.

John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen C. North, and Gordon Woodhull. 2002. Graphviz—open source graph drawing tools. In *Graph Drawing: 9th International Symposium, GD 2001 Vienna, Austria, September 23–26, 2001 Revised Papers 9*, pages 483–484. Springer.

Edmund Grimley Evans. 1997. Approximating context-free grammars with a finite-state calculus. In *Proceedings of the 35th Annual Meeting of the Association for Computational Linguistics and Eighth Conference of the European Chapter of the Association for Computational Linguistics*, pages 452–459.

Marjan Ghazvininejad, Xing Shi, Jay Priyadarshi, and Kevin Knight. 2017. Hafez: an interactive poetry generation system. In *Proceedings of ACL 2017, System Demonstrations*, pages 43–48, Vancouver, Canada. Association for Computational Linguistics.

Kyle Gorman. 2016. Pynini: A Python library for weighted finite-state grammar compilation. In *Proceedings of the SIGFSM Workshop on Statistical NLP and Weighted Automata*, pages 75–80, Berlin, Germany. Association for Computational Linguistics.

Mans Hulden. 2005. Finite-state syllabification. In *International Workshop on Finite-State Methods and Natural Language Processing*, pages 86–96. Springer.

Mans Hulden. 2009a. *Finite-state machine construction methods and algorithms for phonology and morphology*. Ph.D. thesis, The University of Arizona.

Mans Hulden. 2009b. Foma: a finite-state compiler and library. In *Proceedings of the Demonstrations Session at EACL 2009*, pages 29–32, Athens, Greece. Association for Computational Linguistics.

Mans Hulden. 2017. Rewrite rule grammars with multitape automata. *Journal of Language Modelling*, 5(1):107–130.

Mans Hulden. 2022. Finite-state technology. In *The Oxford Handbook of Computational Linguistics, 2nd ed.* Oxford University Press, Oxford.

Ronald M. Kaplan and Martin Kay. 1994. Regular models of phonological rule systems. *Computational Linguistics*, 20(3):331–378.

Lauri Karttunen. 1993. Finite-state lexicon compiler. Technical Report ISTL-NLTT-1993-04-02, Xerox Palo Alto Research Center.

Kevin Knight and Jonathan Graehl. 1998. Machine transliteration. *Computational linguistics*, 24(4):599–612.

Krister Lindén, Miikka Silfverberg, and Tommi Pirinen. 2009. Hfst tools for morphology–an efficient open-source package for construction of morphological analyzers. In *State of the Art in Computational Morphology: Workshop on Systems and Frameworks for Computational Morphology, SFCM 2009, Zurich, Switzerland, September 4, 2009. Proceedings*, pages 28–47. Springer.

Sarah Moeller, Ghazaleh Kazeminejad, Andrew Cowell, and Mans Hulden. 2019. Improving low-resource morphological learning with intermediate forms from finite state transducers. In *Proceedings of the 3rd*

Workshop on the Use of Computational Methods in the Study of Endangered Languages Volume 1 (Papers), pages 81–86, Honolulu. Association for Computational Linguistics.

Mehryar Mohri. 2009. Weighted automata algorithms. In Manfred Droste, Werner Kuich, and Heiko Vogler, editors, *Handbook of Weighted Automata*, pages 213–254. Springer.

Mehryar Mohri and Mark-Jan Nederhof. 2001. Regular approximation of context-free grammars through transformation. In Jean-Claude Junqua and Gertjan van Noord, editors, *Robustness in Language and Speech Technology*, pages 153–163. Springer Netherlands, Dordrecht.

Mehryar Mohri, Fernando Pereira, and Michael Riley. 2008. Speech recognition with weighted finite-state transducers. *Springer Handbook of Speech Processing*, pages 559–584.

Saliha Muradoglu, Nicholas Evans, and Hanna Suominen. 2020. To compress or not to compress? a finite-state approach to Nen verbal morphology. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics: Student Research Workshop*, pages 207–213, Online. Association for Computational Linguistics.

Jean-Eric Pin. 1998. Tropical Semirings. In J. Gunawardena, editor, *Idempotency (Bristol, 1994)*, Publ. Newton Inst. 11, pages 50–69. Cambridge Univ. Press, Cambridge.

Emmanuel Roche and Yves Schabes. 1997. *Finite-state language processing*, volume 115. MIT press, Cambridge, MA.

Ivan A Sag, Ronald Kaplan, Lauri Karttunen, Martin Kay, Carl Pollard, Stuart M. Shieber, and Annie Zaenen. 1986. Unification and grammatical theory. In *Proceedings of the West Coast Conference on Formal Linguistics*. Cascadilla Press.

Daniel Swanson and Nick Howell. 2021. Lexd: A finite state lexicon compiler for non-suffixational morphologies. *Multilingual Facilitation*, pages 133–146.