Lexd: A Finite-State Lexicon Compiler for Non-Suffixational Morphologies

Daniel Swanson[1] and Nick Howell[2]

[1] Swarthmore College, Swarthmore PA, USA awesomeevildudes@gmail.com
[2] HSE University, Moscow, Russian Federation nhowell@hse.ru

Abstract This paper presents `lexd`, a lexicon compiler for languages with non-suffixational morphology, which is intended to be faster and easier to use than existing solutions while also being compatible with other tools. We perform a case-study for Chukchi, comparing against a hand-optimised analyser written in `lexc`, and find that while `lexd` is easier to use, performance remains an obstacle to its use at production level. We also compare performance between `lexd` and `hfst-lexc` for three analysers still in the prototype phase, finding that `lexd` is at least as fast, sometimes faster, to compile; we conclude it is a reasonable choice for prototyping new analysers. Future work will explore how to move `lexd` performance toward production-grade.

Keywords: lexicon compiler · nonlinear morphology · morphotactic transducer · hyperminimisation · finite state morphology.

1 Introduction

This paper introduces `lexd`, a finite-state lexicon compiler which makes development of morphological analysers easier, particularly for non-suffixational morphologies, but at some cost in runtime efficiency.

Finite-state morphological analysis continues to be important for natural language tasks in lesser-resourced languages. Lack of large corpora significantly impede current purely statistical methods, and lack of a large monied speaker base suggests that even newer techniques (e.g. transfer learning) may be slow to bring to market.

Modern finite-state morphology systems feature prominently in the Divvun software built on the Giellatekno research project [11], providing resources for North Sámi, among others; these are based on the free and open-source Helsinki Finite-State Tookit `hfst` [10].

Finite-state morphology is also used in the Apertium machine translation platform targeted at low-resource languages, [6]; the Apertium project uses both `hfst` and their own finite-state toolkit, `lttoolbox` [12]. In addition to the framework, Apertium provides machine translation systems between many pairs of languages, principally pairs which are closely-related.

Finite-state morphology systems are not always easy to develop, however; see [14] for some common complaints. The complaint that development is non-incremental is especially true in non-suffixational morphologies.

Currently, there are two strategies used to deal with such languages in finite-state systems: explicit listing of forms, or over-generating and then adding constraints.

The overgenerate-and-restrict strategy with finite state algebra operations frequently results in explosive growth; a technical solution called *flag diacritics* (see section 2.2) is a popular alternative. Flag diacritics are fast to compile at some runtime performance cost; for non-suffixational morphologies, however, these must be written by hand.

The `hfst` supports flag diacritics, but Apertium's `lttoolbox` does not; further, for use in non-suffixational morphologies, flag diacritics must be carefully written by hand by experienced designers.

We introduce `lexd`, a new lexicon compiler designed to ease the development of high-performance non-linear finite-state morphology systems. In section 2 we review the place and history of lexicon compilers in natural language processing (2.1–2.2), and present the design features of `lexd` in this context (2.4 –2.6). Section 3 gives an overview of our implementation choices and then describes several techniques used to ensure that `lexd` is ready for use. Section 4 describes a case study reimplementing the challenging portions of a morphological analyser for Chukchi (ISO-869-3 `ckt`); `lexd` provides a more natural framework for expressing the morphology of Chukchi, but there is more work to be done to optimise the compiler.

Section 5 gives experimental results showing that for initial prototyping, `lexd` is faster and smaller during compilation (compilation time and memory use), and sometimes in the resulting transducer (transducer lookup performance and size). Finally, section 6 reviews the current state and future work for `lexd` and enabling non-linear finite-state morphology in general.

2 Design

2.1 Review of lexicon compilers

Lexicon compilers provide a framework for finite-state morphology system developers to abstract grammatical patterns; popular lexicon compilers include implementations of the `lexc` source format [3] and the `lttoolbox` dictionary compiler `lt-comp`.

The `lexc` source format [7] describes a tree-like structure in which the root represents the beginning of a string pair (analysis and surface) and each node a new "continuation" of the pair. Naïvely, then, `lexc` can only represent suffixational morphologies.

The XML-based format used by `lt-comp`, meanwhile, builds "paradigms", each of which consists of some number of paths, each of which can contain references to earlier paradigms. While this provides some convenience to the lexicon author, paradigms which are neither initial nor terminal are duplicated at compliation time. The resulting transducer is equivalent to a `lexc`-compiled transducer in which the duplication is performed by the author. Paradigms in `lt-comp` cannot refer to unseen paradigms (for example, defined later in the file); this forbids paradigm cycles. (The equivalent operation, lexicon cycles, is permitted in `lexc`.)

One strategy for dealing with non-suffixational morphologies is to overgenerate, that is, to have every entry point to all continuation lexicons that it ever occurs with. This will result in a small transducer which includes all correct paths, but also includes a number of incorrect ones. For an example, see Figure 1.

Overgeneration must be compensated for; a very general strategy for this is using post-composition restriction transducers, e.g. two-level rule [9] transducers compiled by a `twolc` implementation (the canonical being in [3]).

However, such pure finite-state algebra strategies tend to result in an explosion in transducer size as every entry which has different continuations in different contexts gets duplicated to avoid spurious paths. See Figure 1 for an illustration.

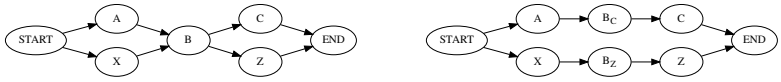


Figure 1. The issue with purely finite-state approaches to non-suffixational morphology. The desired paths are `A B C` and `X B Z`. In the transducer on the left, we overgenerate, producing the undesired paths `A B Z` and `X B C`. In the transducer on the right, meanwhile, we duplicate the continuation lexicon `B`, which can be problematic if `B` is large.

2.2 Flag diacritics and hyperminimisation

A more sophisticated strategy is the use of flag diacritics, special symbols which tools interpret as epsilon transitions. They add a small amount of memory to the path lookup mechansim: paths which contain incompatible flags are discarded early; this requires lookup tooling support.

Flag diacritics can be inserted in continuation lexicons which overgenerate; tools which support flag diacritics will reject the undesired paths when performing lookups in the compiled transducer.

The implementation of flag diacritics in [3] requires manual insertion by the language designer; as a somewhat technical and delicate task, it is desirable to automate this, a process known as "hyperminimisation". For a study on the effects of hyperminimisation and its introduction into `hfst-lexc`, see [4].

The `lexd` compiler implements several different strategies for hyperminimisation, trading off between transducer growth and lookup overhead; see section 3.1 for details.

2.3 Multi-character symbols

Unlike other lexicon compilers, `lexd` does not require the user to explicitly declare multi-character symbols. Instead, we take an opinionated view and design according to the Apertium convention: the `lexd` parser automatically interprets strings in angle brackets or curly braces as multi-character symbols. Unicode characters consisting of multiple codepoints are also automatically encoded as multi-character symbols when appropriate, see 3.2.

This choice restricts the variety of multi-character symbols available; if other forms are necessary (for example, for compatibility with other tooling), they can be transformed via composition.

2.4 Patterns vs. continuations

The `lexd` source format replaces continuation lexicons with "patterns"; a pattern is a named list of entries, and each entry consists of sequence of patterns or lexicons to concatenate. Thus while `lexc` continuations permit branching only at the end of an entry, `lexd` patterns permit branching anywhere.

Whereas the other lexicon formats discussed in 2.1 directly correspond to the branching structure of the underlying transducer, the `lexd` format aims to more closely reflect the way such phenomena would be described in more standard linguistic documentation. We hope that this change will make developing the morphotactic logic of morphological analysers more feasible for non-specialists; see [1] for an alternative approach and [13] for discussion of it in-practice.

Compiling such rules purely finite-state theoretically requires either overgeneration or duplicating lexicons any time they appear non-terminally. Since concatenated duplicated lexicons lead to superlinear growth in transducer size, `lexd` uses overgeneration with several hyperminimisation techniques (see section 3.1) to achieve the same simplicity of code with minimal performance impact.

Terms in a pattern entry can have quantifiers `?`, `*`, `+`, expressions can be bracketed using parentheses, and alternated with `|`. Single-entry lexicons can be constructed without a separate declaration by enclosing the entry in square brackets.

Figure 2. Examples of `lexc` and `lexd` source. Both examples produce strings with some positive number of `a`s followed by a single `b`.

```
lexc                          | lexd
------------------------------|-----------------------------
LEXICON Root                  | PATTERNS
A;                            | A+ B
LEXICON B                     | LEXICON A
b #;                          | a
LEXICON A                     | LEXICON B
a A;                          | b
a B;                          |
```

2.5 Slots and non-linear morphology

Patterns provide an elegant format for describing concatenative (though perhaps non-suffixational) morphologies, but it does nothing to handle templatic morphotactics [8]. The `lexd` source format allows lexicons to be *slotted*; slots from each lexicon entry can be woven together; see Figure 3.

2.6 Tags and filtering

Some languages have patterns of irregularity which considerably complicate the design of a morphological analyser. One example is Tsez/Dido (ISO-869-3 `ddo`), where the ergative

```
PATTERNS
C(1) :V(1) C(2) :V(2) C(3) V(2):

LEXICON C(3)
שׁ מ ר
י שׁ ב

LEXICON V(2)
: ָ<v><p3><sg>: ַ
: ֹ<v><pprs>: ֵ
```

Figure 3. Slotted lexicons; each entry in `V` is woven together with each entry of `C` to create the pairs שׁמר<v><p3><sg>:שָׁמַר, שׁמר<v><pprs>:שֹׁמֵר, ישׁב<v><p3><sg>:יָשַׁב, ישׁב<v><pprs>:יֹשֵׁב.

is often irregular, and in some contexts is forbidden (e.g. on masdar nouns). The typical strategy for this is to declare extra lexicons in combinatorial explosion; `lexd` simplifies this with the notion of tagged strings and tag filters.

Filters can be applied to patterns as well; negative filters are applied to each token in a pattern entry, while positive filters are distributed over alternation: at least one token must match a positive filter.

3 Implementation

The `lexd` compiler is written in standard `C++-14`, and has light runtime depedencies: Unicode support is provided by `icu` and finite-state primitives are provided by the `lttoolbox` library [12]. The compiler is separated into a frontend which parses the source code and backend which uses `lttoolbox` to build the transducer; both are written by hand. It is licensed under the GNU General Public License version 3, and contributions are welcomed on Github[3].

3.1 Hyperminimisation strategies

We have implemented several different hyperminimisation strategies in `lexd`; the savings in deduplication must be balanced against the overhead (both processing and transducer size) of flag diacritics. See section 5 for an analysis of the trade-offs of the various strategies.

All our hyperminimisation strategies use flag diacritics in lexicon entries to ensure that paired lexicons do not overgenerate, and in cases where larger lexicons are only referred to by single patterns, this is often sufficient. For more complex cases, there are three further options.

The naïve strategy, absolute hyperminimisation, uses flag diacritics for branching in all cases except cascading tag filters (see section 2.6). The use of flag diacritics for filters

[3] `https://github.com/apertium/lexd`

Figure 4. Masdar nominalisation of verbs in Tsez. On the left, without the use of the `lexd` tags feature, and on the right, with. The `NonErg` ending tends to propagate through the rules, creating unrestricted and `NonErg` variants.

Without tags

```
PATTERN VerbSuffix
Masdar OblCaseNonErg

LEXICON Masdar
<masdar>:>ани

PATTERN OblCase
Erg
OblCaseNonErg

LEXICON Erg
<erg>:>ä

LEXICON OblCaseNonErg
<gen1>:>с
<gen2>:>з
<ins>:>д
```

With tags

```
PATTERN VerbSuffix
Masdar OblCase[-erg]

LEXICON Masdar
<masdar>:>ани

LEXICON OblCase
<erg>:>ä[erg]
<gen1>:>с
<gen2>:>з
<ins>:>д
```

is being explored. This strategy is best when there are patterns which are referred to many times and either the processor is optimized for handling flag diacritics without significant performance impacts or the size of the transducer in memory is more important than processing speed.

A slight relaxation is basic hyperminimisation, in which pattern (see section 2.4) and lexicon branching is done through flag diacritics, but lexicon tag filtering in addition to cascade filters (section 2.6) are duplicated. This is effective in cases where tag filters select mostly non-overlapping portions of the lexicons.

Lexicon hyperminimisation, meanwhile, deduplicates lexicons and lexicon tag filters. Patterns and cascading filters are not deduplicated. This uses fewer flag diacritics than the other two modes and based on our experiments (sections 4 and 5) this seems to be a good balance of transducer size and processing speed in many cases.

Finally, it is also possible to use `lexd` entirely without hyperminimisation or with hand-written flag diacritics.

3.2 Unicode support

The `lttoolbox` framework implements transitions in wide characters; this is a fixed-width 16- or 32-bit (depending on compiler) abstract encoding. Any symbol taking more than a single wide character must be explicitly encoded into the alphabet of the transducer in both `lttoolbox` and `hfst-lexc`; multi-codepoint Unicode characters (e.g. sequences of combining symbols), as well as (when wide characters are 16-bit) characters requiring all 32 bits offered by Unicode, will be split into two symbols.

```
PATTERN DerivableVerbStem
VerbStem Th?
Nouns [[iv]:[iv]] [<consume>:>y] Th?

PATTERN Derived-N
Nouns
DerivableVerbStem[-iv] [<pass>[Ia]:>{ы}ёТ[Ia]]
DerivableVerbStem[-tv] [<act>[III]:>{ы}ӆь[III]]

PATTERN N
Derived-N[Ia]       Case[Ia]
Derived-N[Ib]       Case[Ib]
Derived-N[III]      Case[III]
```

Figure 5. Cascaded filters in Chukchi noun-verb and verb-noun derivation. `Derived-N[Ia]` matches the tag in the anonymous lexicon in the second entry of `Derived-N`, so it will include that line. Similarly for `Derived-N[III]` and the third entry. `DerivableVerbStem[-iv]`, meanwhile, will never include anything from the second entry because of the `[[iv]:[iv]]` anonymous lexicon.

Since explicit declaration of multicharacter symbols is an anti-goal of `lexd`, we use `icu` to read source programs, which are required to be in `UTF-8` encoding, by-character. For language designers wishing to explicitly split a sequence of combining characters, we trim the base character when combined with `SPACE`. The `lexd` parser does not permit splitting a single Unicode codepoint, unlike `lt-proc` and `hfst-lexc`.

3.3 Feature tests

The `lexd` source code ships with 31 feature- and regression-tests; every feature added and bug fixed requires matching tests to be added to the testsuite, helping to document expected-and-actual behaviour.

Tests are run nightly using the Apertium build infrastructure on ten Linux distributions and MacOS 10.15; standards-compliance is tested against the GNU Compiler Collection and `clang`.

3.4 Fuzzing

Additional testing is provided by fuzzing. At present, the fuzzing script generates one million patterns by randomly selecting from the set of all characters that are meaningful in a pattern and the letters A, B, and C. It then attempts to compile the pattern together with lexicons A, B, and C and records whether compilation succeeds or fails and whether any segfaults or other fatal errors occur. We hope to introduce coverage-guided fuzzing in the near future.

Figure 6. Splitting of Unicode characters in Tsez and Hebrew. In Tsez, the ergative appears as a long-vowel ending. In the `twol` rule below, the two codepoints (vowel and combining character) would be treated separately, and the `V:0` rule would delete the a without deleting the diacritic. In Hebrew, meanwhile, vowels are represented by combining diacritics, which leads to lexicon entries containing diacritics without base characters. These are interpreted as single characters because they are immediately preceeded by spaces.

Tsez – Non-splitting

```
# lexd
PATTERN N
# Noun = lemma oblstem
Noun(1)          AbsCase
Noun(1):Noun(2) OblCase

PATTERN OblCase
Erg
# more

LEXICON Erg
<erg>:>ä
```

```
! twol
! omitted: Alphabet, Vowels, MorphBound

Rules

"rule: drop V in V>V2"
V:0 <=> _ :0* MorphBound+ Vowels ;
        where V in Vowels;
```

Hebrew – Splitting

```
PATTERN Verb
:Infl(1) Root(1) Root(2) :Infl(2) Root(3) Infl(1):Infl(3)

LEXICON Infl(3)
<v><impf><p1><sg>:אֲ : ◌ : # surface: alef-segol, space-holam, null
<v><impf><p1><pl>:נְ : ◌ : # surface: nun-hiriq,  space-holam, null

LEXICON Root(3)
א מ ר
ר ע ה
```

4 Case study: Chukchi reimplementation

We reimplemented the Chukchi (ISO-639-3 `ckt`) morphological analyser from [2]. Authors describe a morphological analyser consisting of a `lexc` lexicon with hand-authored flag diacritics plus a `twol` ruleset enforcing phonological rules, the majority of which are expressions of vowel harmony.

Chukchi has fairly rich inflectional and derivational morphology, but one of the key challenges in finite state designs is the Chukchi system of incorporation in which a noun and a verb can be combined to form a new verb.

We reimplemented the nominal and verbal inflection and derivation systems, including incorporation, for Chukchi in `lexd`. The morphophonology, which enforces vowel harmony among several other phonological rules, is left unchanged; `lexd` code replaces only the lexicon component.

We compare from both static (code size, final size of transducer) and performance (timing and memory use during compile and lookup), the latter across several different system configurations.

We also provide a brief coverage analysis: the `lexd` ruleset analyses words which are un-analysable by the `lexc` implementation, despite being a reimplementation of only a subset of the `lexc` version. This supports our claim that using `lexc` with flag diacritics for non-linear morphology is error-prone.

4.1 Methodology

We examine two variants of the reimplementation. Chukchi has an array of word class-changing derivations, and in principle, these can be iterated. Our "basic" reimplementation permits at most single word class-changing derivations, while our "complex" reimplementation permits unlimited iteration. The two models differ trivially in terms of the `lexd` source code, but the increase in computational complexity is quite large.

Code size and transducer size are reported for combined lexicon+morphophonology code length, along with compilation time and maximum memory usage.

Our coverage analysis is performed over the 100K token corpora from [2]; we provide naïve coverage (both forms and tokens), and also a restricted comparison covering only the morphology present in the reimplementation.

4.2 Results and Discussion

The Chukchi reimplementation was completed over the course of three days; the pattern hierarchy was mostly induced from the Chukchi grammar [5], while lexicons were transliterated from the `lexc` source.

It should be emphasised that the basic organisational strategy of authoring a `lexd` lexicon is "transcribe directly from a grammar;" see Figure 7. The final patterns are only required for circumfixes.

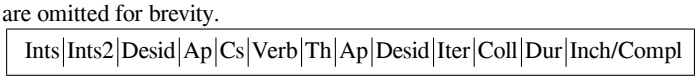
We present static measures in table 1. The majority of compile-time and the high-water mark of memory use both belong to the compose-intersection with morphophonology rules. See section 5 for a more detailed performance analysis of the `lexd` compiler.

Table 1. Code and transducer sizes for Chukchi analysers. Rules are in lines of code, final transducer measured in megabytes, flags in flag diacritic transductions. The remaining measurements are compile time (on an 8-core Xeon E3-1275 v5 running at 3.6GHz) and peak memory usage during compile.

Model	**Rules** (LoC)	**Trans.** (MiB)	**Flags**	**Compile** (CPU-sec)	**Mem.** (GiB)
`lexc`	1.2k	26	105k	86	4
`lexd` (basic)	0.7k	3	133k	485	28
`lexd` (complex)	0.7k	16	1290k	1890	65

Dynamic measures and coverage are provided in table 2. Runtime scales superlinearly with the number of flags used. Memory usage, on the other hand, is significantly

Figure 7. Basic Chukchi verbal derivation; above, as presented in [5] (figure 14.1), and below the `lexd` implementation. Most of the tokens referred to are simple single-entry lexicons; the circumfix derivations `Desid` and `Ap` require subpatterns. Lexicons giving realisations to the various patterns are omitted for brevity.




```
PATTERN VerbalStem
Ints? Ints2? DesidVerbalStem Iter? Coll? Dur? InchCompl?

PATTERN DesidVerbalStem
Desid(1) ApVerbalStem Desid(2)
ApVerbalStem

PATTERN ApVerbalStem
Ap(1) Cs? DerivableVerbStem Ap(2)
Cs? DerivableVerbStem

PATTERN DerivableVerbStem
VerbStem Th?
# other-to-verb derivations
```

improved. We also compute the coverage improvement and loss; coverage improvement of an analyser is the coverage unique to the analyser, and loss is the coverage lacked by the analyser and common to all competitors.

Table 2. Runtime performance for the Chukchi analysers. The 100K token corpus from [2] was distributed over an 8-core Xeon E3-1275 v5 running at 3.6GHz. Measurements are corpus analysis runtime in processor-seconds, peak memory usage, naïve coverage (forms/tokens), and coverage improvement and loss (forms/tokens). The coverage improvement and loss is calculated as follows: `lexc` over `lexd` (complex), `lexd` (basic) over `lexc`, and `lexd` (complex) over `lexd` (basic).

Model	**Analysis** (sec)	**Mem.** (MiB)	**Naïve cov.** (%)	**Cov. imp.** (%)	**Cov. loss** (%)
`lexc`	5	107	11 / 30	+4.0 / 21	−1.0 / 1
`lexd` (basic)	1026	31	7 / 9	+1.0 / 1	−4.3 / 22
`lexd` (complex)	23979	77	8 / 10	+0.7 / 1	−0 / 0

The coverage improvement and loss column shows that the `lexd` model adds mostly rare words to the vocabulary: the ratio of forms to tokens is approximately unity; and that the model lacks high-frequency words, with `lexc` gaining forms to tokens at a ratio of $5 : 1$. Further, we see that the complex `lexd` model almost doubles the coverage improvement of the basic model.

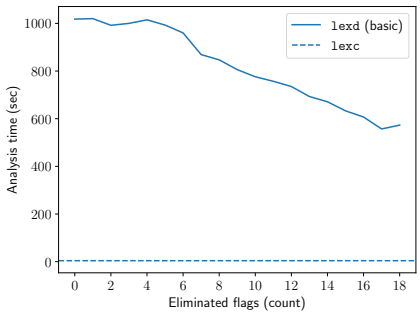
Runtime performance takes an extreme hit; we associate this with the larger number of flag diacritics used by the `lexd` transducers (see table 1). Indeed, before several optimi-

sations to the `lexd` flag diacritic algorithm, authors were unable to complete the analysis. One avenue which unartfully improved the runtime performance was flag elimination.

Eliminating flags involves duplicating some portions of the transducer; this brings an increase in transducer size (both on disk, and memory usage at runtime), but can significantly improve analysis speed. See Figure 8 for the results of elimination on the two implementations.

Our algorithm was to take all flags referenced fewer than 1000 times and incrementally eliminate them. (Eliminating more frequently-referenced flags frequently resulted in the elimination process never terminating, or in terminating due to memory exhaustion.) The order in which they were eliminated was set so that each step produced a transducer minimal among all possible single-flag eliminations.

Figure 8. Performance versus flags eliminated.



5 Performance Analysis

Apertium morphological analysers for Wamesa (ISO-639-3 `wad`), Lingala (ISO-639-3 `lin`), and Navajo (ISO-639-3 `nav`) were converted from `lexc` to `lexd`. These languages represent a variety of non-suffixational morphological phenomena and stages of development. Comparisons of compilation time, memory usage, and runtime efficiency can be found in table 3. To compare runtime efficiency, we used the `lexc` implementation to randomly generate 10000 forms which were then fed into each of the analysers.

5.1 Discussion

In all minimisation strategies, compilation time and memory usage were significantly improved over the `lexc` model. Runtime varies significantly between languages and configurations though either lexicon hyperminimisation or flag diacritics without hyperminimisation is likely to perform reasonably for most applications.

Composition with morphophonological rules is slowed down by epsilon transitions, including flag diacritics, and absolute hyperminimisation doesn't have enough impact on

Table 3. Compilation time, RAM usage, and runtime efficiency for Lingala, Navajo, and Wamesa. Compilation numbers are presented both with and without `twol` rule composition. The best number of each type in each column is bolded. All numbers are the average of 20 runs on a 10-core Intel(R) Core(TM) i9-9900X CPU running at 3.5GHz.

	Lingala	**Navajo**	**Wamesa**
Lexicon size	1700 stems	60 stems	1300 stems
Lexc with path restrictions	270000 ms / 5000 MB	7800 ms / 230 MB	26000 ms / 600 MB
+ twol	270000 ms / 5100 MB	8600 ms / 270 MB	60000 ms / 710 MB
Runtime	69 ms	93 ms	52 ms
Lexd	770 ms / 65 MB	54 ms / **12 MB**	710 ms / 74 MB
+ twol	1200 ms / 120 MB	840 ms / 44 MB	34000 ms / 180 MB
Runtime	**66 ms**	**92 ms**	**46 ms**
Lexd with flags	250 ms / 15 MB	41 ms / **12 MB**	**120 ms / 12 MB**
+ twol	490 ms / 47 MB	**830 ms / 43 MB**	**33000 ms / 95 MB**
Runtime	916 ms	249 ms	64 ms
absolute hypermin.	260 ms / 16 MB	**40 ms / 12 MB**	150 ms / 13 MB
+ twol	530 ms / 51 MB	840 ms / 44 MB	34000 ms / 110 MB
Runtime	10837 ms	825 ms	3073 ms
lexicon hypermin.	**220 ms / 13 MB**	41 ms / **12 MB**	**120 ms / 12 MB**
+ twol	**450 ms / 44 MB**	840 ms / 44 MB	**33000 ms / 95 MB**
Runtime	1371 ms	617 ms	152 ms

the overall size of the transducer in any of these instances to make up for having more flags. Thus it is only the most efficient in term of compilation in the case of Navajo, where the lexicon is small enough that all the three approaches are only marginally different.

6 Conclusion

The new `lexd` source format is capable of naturally describing non-linear morphologies which are challenging to correctly describe using other available systems. At the prototype scale it is not only efficient to to write in and compile, but at runtime as well. Due to the volume of flag diacritics added during hyperminimisation, `lexd` currently is not suitable as a replacement for production-grade hand-optimised flag diacritic-based systems. Improvements to the hyperminimisation system are thus the most important avenue for further research. Strategies currently under exploration include an auxiliary transducer walked in parallel or multi-tape transducers.

There are several new analyser projects which have decided to use `lexd`; the tag-and-filter system, full regular expression-level expressiveness, and refinements of the slot and side syntax have all been implemented to meet their needs. Further feature requests are still in the design stage: lexicon-dictionaries (with named parts and defaults), variables in filtering expressions, additional improvements to tag syntax, and weighting of transducers.

Acknowledgements

The authors would like to extend special thanks to Jonathan North Washington and Francis Tyers for their support and advice in the development of `lexd` and in the writing of the paper. Francis Tyers and Vasilisa Andriyanets also provided critical support in understanding and reimplementing the Chukchi analyser; `lexd` would be far less featureful, useful, or understood without their efforts.

References

1. Alnajjar, K., Hämäläinen, M., Rueter, J.: On editing dictionaries for uralic languages in an online environment. In: Proceedings of the Sixth International Workshop on Computational Linguistics of Uralic Languages. pp. 26–30. Association for Computational Linguistics, Wien, Austria (10–11 Jan 2020), https://www.aclweb.org/anthology/2020.iwclul-1.4
2. Andriyanets, V., Tyers, F.: A prototype finite-state morphological analyser for Chukchi. In: Proceedings of the Workshop on Computational Modeling of Polysynthetic Languages. pp. 31–40. Association for Computational Linguistics, Santa Fe, New Mexico, USA (Aug 2018), https://www.aclweb.org/anthology/W18-4804
3. Beesley, K.R., Karttunen, L.: Finite-state morphology: Xerox tools and techniques. CSLI, Stanford (2003)
4. Drobac, S., Silfverberg, M., Lindén, K.: Automated lossless hyper-minimization for morphological analyzers. In: Proceedings of the 12th International Conference on Finite-State Methods and Natural Language Processing 2015 (FSMNLP 2015 Düsseldorf). Association for Computational Linguistics (2015), https://www.aclweb.org/anthology/W15-4806
5. Dunn, M.J.: A grammar of Chukchi. Ph.D. thesis, The Australian National University (1999), http://hdl.handle.net/1885/10769

6. Forcada, M.L., Ginestí-Rosell, M., Nordfalk, J., O'Regan, J., Ortiz-Rojas, S., Pérez-Ortiz, J.A., Sánchez-Martínez, F., Ramírez-Sánchez, G., Tyers, F.M.: Apertium: a free/open-source platform for rule-based machine translation. Machine Translation (jul 2011). https://doi.org/10.1007/s10590-011-9090-0, http://www.springerlink.com/content/h134p1j73377071k/export-citation/
7. Karttunen, L.: Finite-state lexicon compiler. Tech. Rep. ISTL-NLTT-1993-04-02, Xerox Palo Alto Research Center (1993)
8. Kiraz, G.A.: Multitiered nonlinear morphology using multitape finite automata: a case study on syriac and arabic **26**(1), 77–105 (2000), https://www.aclweb.org/anthology/J00-1006
9. Lauri, K., Kenneth, B.: Two-level rules compiler. Tech. Rep. ISTL-92-2 (1992)
10. Lindén, K., Silfverberg, M., Axelson, E., Hardwick, S., Pirinen, T.: HFST—Framework for Compiling and Applying Morphologies, Communications in Computer and Information Science, vol. Vol. 100, pp. 67–85. Springer (2011)
11. Moshagen, S.N., Rueter, J., Pirinen, T., Trosterud, T., Tyers, F.M.: Open-source infrastructures for collaborative work on under-resourced languages (01 2014)
12. Ortiz Rojas, S., Forcada, M.L., Ramírez Sánchez, G.: Construcción y minimización eficiente de transductores de letras a partir de diccionarios con paradigmas. Revistas - Procesamiento del Lenguaje Natural (35) (2005)
13. Rueter, J., Hämäläinen, M., Partanen, N., et al.: Open-source morphology for endangered Mordvinic languages. In: Proceedings of Second Workshop for NLP Open Source Software (NLP-OSS). The Association for Computational Linguistics (2020)
14. Wintner, S.: Strengths and weaknesses of finite-state technology: A case study in morphological grammar development. Natural Language Engineering **14**, 457–469 (10 2008). https://doi.org/10.1017/S1351324907004676