Parser combinators for Tigrinya and Oromo morphology

Patrick Littell[1], Tom McCoy[2], Na-Rae Han[3], Shruti Rijhwani[4]
Zaid Sheikh[4], David Mortensen[4], Teruko Mitamura[4], Lori Levin[4]
[1]National Research Council of Canada, [2]Johns Hopkins University, [3]University of Pittsburgh, [4]Carnegie Mellon University
Digital Technologies, Dept. of Cognitive Science, Dept. of Linguistics, Language Technologies Institute
1200 Montreal Road, 3400 N. Charles Street, 2816 Cathedral of Learning, 5000 Forbes Avenue,
Ottawa, ON K1A 0R6, Baltimore, MD 21218, Pittsburgh, PA 15260, Pittsburgh, PA 15213
patrick.littell@nrc.gc.ca, tom.mccoy@jhu.edu, naraehan@pitt.edu, {srijhwan, zsheikh, dmortens, teruko, lsl}@cs.cmu.edu

Abstract

We present rule-based morphological parsers in the Tigrinya and Oromo languages, based on a *parser-combinator* rather than *finite-state* paradigm. This paradigm allows rapid development and ease of integration with other systems, although at the cost of non-optimal theoretical efficiency. These parsers produce multiple output representations simultaneously, including lemmatization, morphological segmentation, and an English word-for-word gloss, and we evaluate these representations as input for entity detection and linking and humanitarian need detection.

Keywords: morphology, parsing, Tigrinya, Oromo

1. Introduction

In this paper, we experiment with using parser combinators (Hutton and Meijer, 1988; Frost and Launchbury, 1989) for the rapid development of practical morphological parsers, as an alternative or supplement to the typical finite-state transducers (Karttunen and Beesley, 1992; Karttunen, 1993). This paradigm offered some practical advantages over a finite-state system, allowing parsers to be written very rapidly in a familiar programming language, although at a cost of runtime efficiency.

We present morphological parsers for two Afroasiatic languages, the Tigrinya language of Eritrea and Ethiopia (§4.1.), and the Oromo language of Ethiopia and Kenya (§4.2.).[1] These parsers were designed during the LoReHLT17 "surprise-language" evaluation (Strassel and Tracey, 2016) (§3.) to support machine translation, entity detection and linking, and humanitarian need detection (Strassel et al., 2017). These parsers were operable within about 36 hours of learning the identity of the languages, although they underwent further development during the next two weeks of evaluation.

2. Parser combinators

2.1. Introduction

The "parser combinator" paradigm (Burge, 1975; Wadler, 1985; Hutton and Meijer, 1988; Frost and Launchbury, 1989) is a kind of declarative programming that simultaneously defines the grammar being parsed and the executable code that parses it. This paradigm involves defining parser functions with a particular semantics, using a general-purpose programming language (in our case, Python), as well as defining higher-order "combinators" that take one or more component parsing functions as arguments and return a complex parsing function (for example, the composition of the two functions).[2] The resulting grammar is itself an executable function, that parses text as a recursive-descent parser, but one in which every component has access to all the capabilities and libraries of the general-purpose programming language (e.g. regular expression libraries, file I/O, etc.).

The basic building block of a parser-combinator grammar is the *atomic parser*. An atomic parser can do something as simple as recognize a single letter—in our examples, a parser defined as `Tex("d")` would recognize a single character `"d"` at the edge of the input's `Text` representation. By default, our parsers consume input from the right edge of a word because of most languages' tendency towards suffixation, but this is configurable. Parsers return a set of ⟨*output, remnant*⟩ ordered pairs, where *remnant* is what is left over from the input (e.g., the string without `"d"`), and *output* is any arbitrary Python object, typically some manner of structured or augmented representation of what was parsed. The return value of a parser is defined as a set, rather than a single ordered pair, because there may be, at any particular stage of the parse, multiple competing hypotheses regarding what the output representation should be and what remains to be parsed.

In our implementations, we typically define one atomic function to parse one morph, rather than define atomic parsers for individual characters. We also make significant use of generators, trivial atomic parsers that consume no input, but put a textual representation (such as a gloss) into one of several output channels. The concatenation of parsers with generators achieves the transduction between multiple different types of representation.

The other kind of parsers in this paradigm are *parser combinators*, functions that take one or more parsers as input and return a parser as output. The two prototypical parser combinators are concatenation and disjunction (implemented in our system by overloading Python's `+` and `|` operators, respectively). A concatenation `A+B` is defined as applying the parser A to each remnant produced by B and concatenating the output representations[3]; a disjunction `A|B` is simply

[1]`github.com/littell/ethi_morph`

[2]The usage of "combinator" here comes not from combinatoric logic but from the "combinator pattern" (`wiki.haskell.org/Combinator_pattern`) of software design.

[3]As noted above, our parsers default to right-to-left parsing,

```
> ROOT = Tex("jump") | Tex("talk") | Tex("think")
> SUFFIX = Tex("ed") | Tex("ing") | Tex("s") | NULL
> WORD = ROOT + SUFFIX
> WORD.parse("jumping") != [] # Does the parser return any results?
True
> WORD.parse("talker") != []
False
```

Figure 1: A simple English parser for twelve inflected verb forms, illustrating how concatenation and disjunction combinators allow the executable definition of a parser to resemble a familiar BNF-like grammar specification. Note that in a real grammar, we would not list every root or stem (e.g., "jump") as a literal expression in the code; rather, we would typically define a "lookup" parser that loads in one or more dictionary files from disk.

```
> ROOT = Tex/Mor/Lem("ugaandaa") + Glo/Nat("Uganda")
> SUFFIX = Tex/Mor("tti") + Glo("LOC") + Nat("in (.*)")
> WORD = ROOT + SUFFIX
> WORD.parse("ugaandaatti")
[{"breakdown":"ugaandaa-tti", "lemma":"ugaandaa", "gloss":"Uganda-LOC", "natural":"in Uganda"}]
```

Figure 2: A simple Oromo example parser with multiple output representations: a morphological breakdown, a lemma, a glossed breakdown, and a more naturalistic English-like gloss.

defined as the union of the results.

By overloading + and | in this way, we can express the *code* that executes as if it were the *grammar* that it is parsing. That is to say, a formal representation of the grammar is also the actual code that executes to parse this grammar; Figure 1 illustrates this for a few English verbs.

2.2. Example

The parsers described in this paper use a combination of parser function objects (which consume input strings) and generator function objects (parsers that trivially succeed without consuming input, while outputting additional representations) to convert one representation into others. Types of representation are conceptualized in this system as "channels", from which the parsers consume or output text representations. For example, in Figure 2, there are five channels, an input channel `Tex` (text) and four output channels: `Mor` (morphological breakdowns), `Lem` (lemmas), `Glo` (glossed breakdowns), and `Nat` (naturalistic English-like glosses).

A textual representation (like the suffix `"tti"`) is turned into a parser or generator by associating it with a specific channel: `Tex(X)` defines a parser that consumes `X` from the `Tex` input channel, while `Mor(X)` defines a generator that consumes no input but outputs `X` into the `Mor` output channel. Where the representations happen to be identical, as they are in this example for the suffix `"tti"`, this can be abbreviated as `Tex/Mor(X)`; this is just syntactic sugar for `Tex(X) + Mor(X)`.

In this Oromo example, the surface form `ugaandaatti` ('in Uganda') is broken down and transformed by a concatenation of parsers (the `Tex` components) and generators (the `Mor`, `Lem`, `Glo`, and `Nat` components). The `Tex` parsers consume the suffix `"tti"` and the root `"ugaandaa"` in turn; while the `Mor` generators consume no input but put `"tti"` and `"ugaandaa"` into the appropriate output channel (with the formatting appropriate for that channel, in this case a hyphen). The nature of concatenation is such that if any of these components fails to produce an $\langle output, remnant \rangle$ pair, the whole will fail to produce any output, so the `Mor`-channel generation only survives if the `Tex`-channel parsing succeeds.

Meanwhile, the `Glo` components generate `"LOC"` and `"Uganda"`, the `Lem` component only generates `"ugaandaa"`, and the `Nat` components produce a naturalistic gloss by generating `"Uganda"` and also inserting it into a particular template (`"in (.*)"`).

Since these parsers can return multiple outputs, the outputs were ranked heuristically by adding penalties for generating lemmas that were not found in any of the available dictionaries, for parses found in the dictionary but with unlikely (according to an English language model) definitions, and for parses that contain certain dispreferred morphemes.

2.3. Advantages of parser combinators

Parser combinators have several practical advantages, particularly in time-constrained situations:

Familiar grammar format The morphological grammars have a familiar, Backus-Naur-like format that mirrors the way linguists already think about grammars. Unlike grammars defined in terms of continuation classes, these grammars are easy to refactor as the linguists/programmers discover more about the morphotactics of the language in question.

For example, when one discovers that tense suffixes do not immediately follow verb roots, but there is a mood suffix that can intervene, one does not have to update every verb root so that its continuation class is `MoodSuffix` rather

so B is evaluated first unless the programmer specifies otherwise. Also, what "concatenation" means depends on the type of output representation; often, it is just a string concatenation of the output strings with a delimiter.

than `TenseSuffix`; one simply has to change `VerbRoot + TenseSuffix` to `VerbRoot + MoodSuffix + TenseSuffix` on one line of the grammar.

Familiar programming syntax and environment The programming syntax and execution environment is familiar Python. Boilerplate and repetitive code (e.g., a class of morphemes all of which have a complex environmental restriction or cause a particular morphophonological change) can be automated within the code itself; it is unnecessary to have a separate transpilation or pre-processing step, as was done in (Littell et al., 2014), to enable a new command or syntactic sugar. Even complex functions entirely outside of the parsing paradigm (e.g., orthographic conversion and normalization, dictionary lookup, etc.) can be wrapped up as a parser object and integrated into the morphological grammar.

Multi-output parsers It is straightforward to associate a parser with multiple different kinds of outputs (Hutton, 1992) letting us simultaneously write parsers that target different representations for different NLP tasks.

Intuitive representation of morphological phenomena Some morphological phenomena that are awkward to express as finite-state transducers are more straightforwardly expressed in a recursive grammar, such as the kinds of templatic or circumfixal morphology that requires finite-state transducers to be extended with flag "memory" (Pretorius and Bosch, 2003; Bower et al., 2017).

It is worth noting that there is no conceptual requirement that an atomic parser define a string truncation like "remove `'d'` from the end of a string", although because of the concatenative nature of most morphology this is the most common kind of atomic parser. The relationship between the input string and the remnant string can be any string-to-string transduction. In the Tigrinya system (§4.1.), we defined some parsers using regular expression substitutions, to handle some particularly difficult plurals that involve both reduplication and root-and-pattern morphology.[4]

Ease of extension We should emphasize here that the above advantages are not just put forward as benefits of a particular parsing *library*, but of a programming technique; one of the benefits of mastering this technique is that, because parser combinators are themselves simple to write from scratch, the programmer is not constrained by the capabilities of an existing parsing library. Extending a library—or rewriting it entirely—is often only a matter of an hour or two, and is not a separate process from programming the grammar itself, since both the library and the grammar are written in the same programming language.

Nevertheless, the small parser-combinator library that we release with these Tigrinya and Oromo grammars should serve as a good starting point for the development of morphological parsers in other languages, as it includes a number of convenience features for the particular problem domain, like predefined output channels for typical word representations (e.g. morphological breakdowns and glosses), specialized parsers for root-and-pattern morphology and reduplication, and combinators that allow parsing either from the left (for prefixes) and the right (for suffixes).

2.4. Disadvantages of parser combinators

On the other hand, there are some drawbacks compared to finite-state systems:

Efficiency After compilation, a finite-state transducer executes in linear time, while parser combinators result in a recursive descent parser with potentially exponential time complexity. For the most part, *morphological* grammars do not have the kind of complexity (in particular left-recursion) that leads to worst-case performance, but nonetheless it is important to note that the responsibility for parser performance here falls back onto the programmer, rather than being handled in the compiler, which is a clear benefit to the finite-state paradigm.[5]

Multi-representation ambiguity While the ability of our parser combinators to define relationships between multiple levels of representation (e.g. text, breakdown, gloss, etc.) was practically useful in a multi-task setting like LoReHLT17, the ambiguity in *each* representation is multiplicative with others. For example, if a particular word has five possible parses in one representation, three in another, and two in another, the parser could return as many as thirty representations. This can pose an efficiency problem, since parser combinators do not have the inherent efficiency of finite-state systems when faced with parse ambiguity.

This is a downside of the particular multi-representation system that we engineered here, but more broadly the arbitrary complexity of *outputs* in a parser-combinator system (which we leveraged here to allow multiple output representations) present another possible source of inefficiency that is not a concern in two-level finite-state systems.

3. LoReHLT17

Our parsers were constructed in the context of the 2017 Low Resource Human Language Technologies (LoReHLT) evaluation[6]. LoReHLT takes the form of a "surprise language" exercise (Oard, 2003), in which competitors are asked to produce machine translation, entity detection and linking (EDL), humanitarian need detection, and sentiment detection in one or more low-resource languages within a series of timed checkpoints, without knowing ahead of time what the languages will be. LoReHLT17 had its first checkpoint after 3 days and two additional checkpoints after 10 and 17 days; future LoReHLT evaluations will have their first checkpoint after 24 hours. In this timeframe, any hand-written rule-based systems must prioritize *programmer time* along with runtime efficiency: they must be the kind of systems that can be written in a day or two.

When a team writing morphological analyzers is asked to support more than one task such as MT or NER/EDL, it is not unusual for the consumers (which might include both

[4]Note, however, that allowing parsers to execute arbitrary transductions removes some guarantees—it is possible to define parsers that never halt—and some possibilities for optimization.

[5]In the time-constrained environment of LoReHLT17, however, the training and test corpora were sufficiently small that runtime efficiency was not the primary bottleneck; programmer time was a more pressing concern, especially during the earliest stages.

[6]`www.nist.gov/itl/iad/mig/lorehlt17-evaluations`

```
> NTexMor = lambda x : Tex/Mor(x) | Truncate("t", Tex) + Tex("n") + Tex/Mor(x)
> ROOT = Tex/Mor/Lem("nyaat") + Glo/Nat("eat")
> SUFFIX = NTexMor("na") + Glo("1PL.PRS") + Nat("we (.*)")
> WORD = ROOT + SUFFIX
> WORD.parse("nyaanna")
[{"breakdown":"nyaat-na", "lemma":"nyaat", "gloss":"eat-1PL.PRS", "natural":"we eat"}]
```

Figure 3: An Oromo example parser illustrating the phonological process that `t` $\rightarrow$ `n` | _`n`. The `NTexMor` line encodes this phonological rule, and writing a morpheme as an argument to `NTexMor`, as with `NTexMor("na")`, indicates that the rule applies at boundaries involving that morpheme.

NLP systems and human annotators) to request different representations of the morphology. Does the team have to write three or four different parsers or *reparsers* to parse outputs into different formats? Our approach makes this unnecessary, providing a good balance between programmer time and output flexibility.

4. System description

This year's LoReHLT task involved Tigrinya and Oromo, two languages spoken in the Horn of Africa. In this section we describe the parser-combinator systems that we created for these languages. These two case studies illustrate some of the benefits of the parser-combinator framework in the context of a time-sensitive task.

4.1. Tigrinya

The Tigrinya language is a member of the Semitic branch of the Afroasiatic family. It is spoken by over 7 million people, mainly in Ethiopia and Eritrea. [7] Like many other Semitic languages, Tigrinya has a templatic morphology system, meaning that the surface form of a word's root morpheme differs in different morphological contexts. For example, the lexical entry for 'bee' is just the consonants */nhb/*, and the vowels appearing between these consonants differ for different inflections of this root, such as the singular form *[nɨhɨbi]* and the plural form *[ʔanahɨb]*. Such root-template patterns are nonconcatenative, meaning that they cannot be expressed simply as the concatenation of morphemes. Because finite-state transducers are inherently concatenative mechanisms, nonconcatenative processes pose a special challenge to finite-state methods. Below we will describe how our system overcomes this challenge as an example of how our parser combinator framework solves some issues inherent in using finite-state methods.

Tigrinya is written in the Ge'ez script which is a form of alphasyllabary (abugida). We begin by converting the script to IPA using Epitran[8], a Python library for transliterating orthographic text as IPA (International Phonetic Alphabet). We used two Epitran mappings for Tigrinya. The first is Epitran's `tir-Ethi` mapping, which is a faithful one-to-one transliteration wherein each Ethiopic symbol is realized as a consonant-vowel sequence (e.g., ትግርኛ → *tɨgɨrɨɲa*). There is, however, an ambiguity in one set of letters (the "sixth series"), which represents both consonants followed by /ɨ/ (e.g. /gɨ/) and consonants that are not followed by any vowel (e.g. /g/). In this first Epitran mapping, all such letters are transliterated as *Cɨ* unless they are at the end of the word, in which case they are transliterated as just *C*.

The second mapping is `tir-Ethi-pp`, a 'precision-phonemic' IPA representation (e.g., ትግርኛ → *tɨgrɨɲa*). In this system, sixth series letters are realized as consonants and /ɨ/ is inserted where demanded by the syllable structure, yielding a phonemic representation that is closer to Tigrinya speech and more suitable for some tasks downstream from our morphological analyzer, including speech recognition. Keeping to the `tir-Ethi` representation system-internally enabled us to simplify the grammar development process where the success of morphological analysis is defined on the level of orthographic, rather than phonetic, tokens, with the exception of the word-final position where the final vowel, present in the script, is omitted.

Grammar development centered around several different linguistic aspects. Concatenative morphology was the initial focus leading up to checkpoint 1. Plural suffixes, negative prefixes and pronominal clitics on verbs were handled, while minimal attention was paid to morpho-phonemic processes. At the same time, a frequency list of Tigrinya word types was compiled from a corpus, which served as (1) a type list to prioritize, and (2) in the absence of gold data, a basis for recall-oriented performance evaluation and monitoring.

Tightening up morpho-phonemic rules for better handling of allomorphs and treatment of templatic verbal morphology became the main goals for the second checkpoint. However, documentation on numerous templates was incomplete at best. Given this lack of information and time pressure, we decided to look to existing solutions. Gasser's HornMorpho (Gasser, 2011) is a finite-state transducer (FST) capable of analyzing Tigrinya verbs (but not nouns) with a reported 96% accuracy. We compiled a list of the 5,700 most frequent verb types, processed them through HornMorpho, re-parsed the output to conform to our system's output, and made the cached analyses available to the lookup routine.

Before the final checkpoint, we concentrated on the so-called "internal plurals" – plural nouns built from consonantal roots and templates. They were handled via several regular-expression-based root patterns, 8 sets in total, extending the system's coverage to such plural nouns as ኣናህብ *ʔanahɨb* 'bees', whose singular form is ንህቢ *nɨhɨbi* 'bee'. To handle root-and-template patterns, we expressed these templates as Python regular expressions, and wrapped

[7] According to Ethnologue: `https://www.ethnologue.com/language/tir`.

[8] `github.com/dmort27/epitran`

each into a parser function; this way non-concatenative morphology and concatenative morphology could be combined.

The class of nouns for the ‘bee/bees’ example above used the following template, which operates on a triplet of the pre-defined consonant class `C`, which are integrated into the specified positions into a regular expression

```
pat2 = 'ʔa(%s)a(%s)ɨ(%s)' % (C, C, C)
```

Additionally, the parser’s lexicon base was dramatically expanded to include a gazetteer, hand-edited entries, and multiple dictionaries sourced from the web and known multilingual resources.

As stated earlier, coverage of the morphological system as measured against a Tigrinya corpus was the only metric of performance available to us, and each update to the system was made sure to result in an increase in recall. At the same time, we closely monitored the system’s performance on the annotation front, with annotators reporting in anomalies or undesirable parser behaviors.

Of particular importance was achieving a balance between coverage and overanalysis. An example would be place names such as ኡጋንዳ *ʔuganɨda* ‘Uganda’: unless they are present in the lexicon, such proper nouns will be overanalyzed, with the final ‘a’ separated out as the 3rd person singular feminine possessive marker. With an abundance of country and place names ending in ‘a’, such overanalysis would have a negative impact on downstream applications such as entity detection and linking. We addressed this issue by employing the `Cost` channel of the parser combinator library, which allows the programmer to add an arbitrary penalty anywhere in the grammar. The total penalty associated with a particular parse output is then taken into account by the heuristic disambiguator.

In the case of ‘a’, the suffixation rule was penalized with a sufficient amount of cost to make sure that the analyzed root+’a’ form gets outranked by the whole, unanalyzed word, unless the remnant root is found in the lexicon. Many similar affixation rules were penalized with varying amounts of cost, which were manually determined and assigned with the aim to engineer desired ranking behaviors among multiple analysis candidates.

4.2. Oromo

Oromo is a language in the Cushitic branch of the Afroasiatic language family. It is spoken by nearly 25 million people in Ethiopia. [9] Unlike Tigrinya, Oromo does not possess templatic morphology. Instead, its morphological processes are generally concatenative, which means that they involve only the concatenation of stems and affixes (though it also has several phonological changes that occur at morpheme boundaries, meaning that its morphological processes do not consist of plain concatenation). These affixes specify many different features including tense and aspect (for verbs) or gender and number (for nouns).

Because Tigrinya morphology poses certain challenges to traditional finite-state methods, we used the Tigrinya section to showcase a theoretical advantage of our system in terms of its expressive capabilities. Oromo morphology, on the other hand, does not pose such theoretical challenges, so we will instead use this section to illustrate the practical benefits of our parser combinator framework. That is, even for a case (such as Oromo) where the increased expressive capabilities of parser combinators are not a benefit, we show how the ease of implementation of parser combinators can make them a good choice for implementing morphological parsing under time constraints.

Oromo uses the Roman script, which is sufficiently close to a phonological representation of the language that we did not have to transliterate Oromo into IPA as was necessary for Tigrinya. For checkpoint 1, we focused mainly on nominal morphology because nouns are the most crucial part of speech for some of the downstream tasks such as NER-EDL. For example, Figure 2 identifies the lemma *ugaandaa* within the word *ugaandaatti*, informing the downstream NER system that the two words refer to the same named entity. For the next checkpoint, we added handling of other parts of speech to the parser, with a focus on verbs (since most multi-morpheme words in Oromo are either nouns or verbs). For the final version of the parser included in the submission to checkpoint 3, we polished the existing functionalities of the parser, increased its lexical coverage, and incorporated orthographic normalization.

As stated above, one of the major advantages of using parser-combinators is in saving programmer time. The construction of the Oromo grammar provides case studies in how parser combinators can save programmer time through three of their key properties: integration with Python, multiple channels, and streamlined handling of phonological transformations.

Integration with Python: It is easy to incorporate Python functions and objects into the parsing pipeline. We used a custom Oromo orthography normalizer, written in Python, to handle considerable variability in Oromo orthography such as multiple different spellings for “Ethiopia” (⟨Itoophiya⟩, ⟨Itoophiyaa⟩, ⟨Itophiyaa⟩, etc.). Also, as our set of lexical resources for Oromo grew throughout the project, we added dictionaries including one that was dynamically updated as human annotators annotated text in Oromo. Here, the Python environment made it easy to read in each dictionary and compile it into a Python dictionary for lexical lookup, whereas without the Python IO tools some more tedious method for adding each new word/definition pair to the parsing program would have been necessary.

Multiple channels: Figure 2 shows a simple chunk of Oromo grammar written using parser-combinators. This grammar has four different channels covering morphological breakdown, lemma, gloss, and natural gloss. In a standard finite-state parser, each one of these four channels would have had to be implemented by its own separate finite-state transducer, but with parser-combinators all four can coexist in the same file. Such consolidation is helpful in two ways: First, it saves the programmer time during the initial creation of the grammar. Second, when making future updates to the grammar, the consolidation means that the programmer only has to update one program rather than

[9] According to the 2007 Ethiopian census, available at `http://microdata.worldbank.org/index.php/catalog/2747`.

Language	Measurement	Original word	Lemma
Tigrinya	typed_mention_ceaf_plus F1	0.478	**0.521**
Oromo	typed_mention_ceaf_plus F1	0.355	**0.376**

Table 1: Results on entity detection and linking, before and after adding lemmatization.

Language	Measurement	Original word	Lemma
Tigrinya	SFType F1	0.325	**0.333**
Tigrinya	SFType occurrence-weighted F1	0.422	**0.471**
Oromo	SFType F1	0.047	**0.086**
Oromo	SFType occurrence-weighted F1	0.051	**0.124**

Table 2: Results on Situation Frame detection, before and after adding lemmatization.

four, possibly saving the effort of version control across multiple files. Since our grammars were continuously being updated, streamlining the updating process was very important.

Streamlined handling of phonology: In traditional FSTs, phonological transformations are typically handled with two separate transducers, one that builds up what might be termed an underlying representation of a word (such as *inlogical*) and another that transforms this underlying representation into a surface form (such as *illogical*). This is computationally elegant, but can be difficult to engineer, since it separates the phonological effects of morphemes in code from the specification of the morphemes that trigger those effects, necessitating workarounds (e.g. special, non-pronounced characters) to pass information between the transducers. For example, many phonological transformations only occur at morpheme boundaries, so a finite-state grammar writer has to keep track of morpheme boundaries to make sure that *inlogical* turns to *illogical* but *only* does not turn into *olly*. Phonological transformations can also be sensitive to exactly which morphemes are being combined, so it might be necessary to keep track not just of where the morpheme boundaries are but also what sort of morpheme boundaries they are so that, for example, *inlogical* turns to *illogical* but *unlikely* does not turn into *ullikely*.

For parser-combinators, however, a phonological rewrite rule can be represented simply as a parser and treated like any other parser in the BNF-style rules of the grammar. This removes the need for a separate finite-state transducer to handle the phonology and also removes the need to keep track of different types of morpheme boundaries.[10] Since this approach require no single lexical representation intermediate between orthographic form and morphological form, as is typically the case with FST morphological analyzers, every juncture can be handled on a morpheme-by-morpheme basis. This necessarily results in some loss of generality but is actually a boon for maintainability. Figure 3 shows one Oromo phonological rewrite rule implemented with parser-combinators.

[10]On the other hand, since these rules are themselves complex parsers, they increase the complexity of the grammar and including many of them can affect the runtime performance of the parser.

5. Experiments and results

We report here the results of adding the Tigrinya and Oromo parsers (specifically, their lemmatization function) to our LoReHLT17 entity detection and linking (§5.1.) and humanitarian need assessment (§5.2.) systems.
It should be noted that in all tasks the scores for Oromo are substantially lower than the scores for Tigrinya; all participants in LoReHLT17 encountered this effect, due largely to the relatively small parallel corpora and lexicons available in Oromo and the large amount of spelling variation in Oromo text.

5.1. Entity Detection and Linking

Entity detection and linking (EDL) for LoReHLT17 was concerned with the recognition of named entities (a subset of proper nouns) in text, their categorization as one of four entity types (person, organization, location, geopolitical entity), and their linking to an external knowledge-base of entities (compiled from several existing databases). The primary metric for EDL in LoReHLT17 was *typed_mention_ceaf_plus*, an F1 measure of detecting the entity and getting both the category and the link correct. We used word-to-word translation with bilingual lexicons for linking entities to the knowledge-base (Pan et al., 2017). Adding lemmatization improved translation of the entities and resulted in F1 point gain for both Tigrinya and Oromo, as seen in Table 1.

5.2. Situation Frame detection

Situation frames (SFs) are a structured representation of events intended to "enable information from many different data streams to be aggregated into a comprehensive, actionable understanding of the basic facts needed to mount a response to an emerging situation" (Strassel et al., 2017). Situation frame detection involves detecting eight humanitarian requirements (e.g. water, food, medicine, evacuation) and three background issues (e.g. terrorism or civil unrest), linking these needs and issues to places, and determining whether or not this is a current, urgent, unrelieved need.
The basic evaluation metric for SF detection in LoReHLT17 is SFType[11]—whether the frame identify the correct

[11]Additional metrics, such as SFType+Location, evaluate whether both the type and other fields of the frame are correct. These are bounded from above by SFType, and our improvement

needs and issues—measured by mean F1 and occurrence-weighted F1, which differ according to whether or not a situation frame is given greater weight when multiple annotators have annotated it. (That is to say, if only one annotator detects an evacuation need in a document, it counts less towards occurrence-weighted F1 than if all annotators detected it.)

Table 2 shows the results of adding lemmatization to our keyword-based situation frame detection system, compared to a system that attempts to identify keywords without lemmatization. Lemmatization adds a ~.01 F1 point improvement (~.05 when weighted for occurrence) to the Tigrinya system, and roughly doubles Oromo performance with a ~.04 F1 point improvement (~.07 when weighted for occurrence).

6. Future research

While this paper has presented parser combinators as if they were in opposition to finite-state methods, the two paradigms are compatible; the ability of parser combinators to incorporate arbitrary functions into their parsing paradigm means that there are no conceptual reasons why some parts of the grammar could not be parsed in a finite-state manner and others in a recursive-descent manner. We are therefore looking into the possibility of integrating Foma FSTs (Hulden, 2009) as parser functions, and/or compiling "safe" subgraphs of the grammar into finite-state systems, to take advantage of the linear time execution where it is possible.

The other benefit of finite-state parsers is that they can be run "backwards" (that is, generating rather than parsing). Incorporating this ability into a parser-combinator framework would be valuable both for pure parser-combinator systems and for the hybrid systems proposed above. The small parser combinator library released with these parsers already supports this to a limited degree: as seen in the examples in §2., the syntax for expressing a parser (like `Tex("tti")`) and a generator (like `Mor("tti")`) is identical, and a subset of parsers/generator functions have implementations such that they can either parse or generate depending on what channel is considered to be the input. A parser that consists solely of these functions can parse and generate in any direction (that is, between *any* two representations that the parser supports); however, both implementations described here use parser functions that do not yet have corresponding generators. We intend, in further development, to augment the library such that all parsers have a corresponding generator and thus any grammar written with this library can parse/generate between all of its representations.

7. Conclusion

By utilizing a declarative programming paradigm that allowed our linguist-programmers to use a familiar grammar formalism within a familiar general-purpose programming language, we created Tigrinya and Oromo parsers within a limited time-frame, that nonetheless led to consistent improvements in entity detection and linking and humanitarian need detection.

These parsers allowed us to rapidly capture some types of morphology (in particular root-and-pattern morphology) that, although possible within finite-state systems, can be difficult to engineer. In future work, we plan to generalize these specialty parsers into a general framework for parsing morphology-specific phenomena. We also intend to make wrappers that allow interoperation with finite-state systems, so that the efficiency of finite-state techniques can be combined with the ease of engineering of parser-combinator techniques.

on these metrics by adding lemmatization are roughly proportional to the SFType improvements shown in Table 2.

8. Acknowledgements

This project was sponsored by the Defense Advanced Research Projects Agency (DARPA) Information Innovation Office (I2O), program: Low Resource Languages for Emergent Incidents (LORELEI), issued by DARPA/I2O under Contract No. HR0011-15-C-0114.

9. Bibliographical References

Bower, D., Arppe, A., Lachler, J., Moshagen, S., and Trosterud, T. (2017). A morphological parser for Odawa. In *Proceedings of 2nd Workshop on Computational Methods for Endangered Languages (ComputEL-2)*.

Burge, W. H. (1975). *Recursive Programming Techniques*. Addison Wesley.

Frost, R. and Launchbury, J. (1989). Constructing natural language interpreters in a lazy functional language. *The Computer Journal*, 32:108–121.

Gasser, M. (2011). HornMorpho: a system for morphological processing of Amharic, Oromo, and Tigrinya. In *Proceedings of the Conference on Human Language Technology for Development*, Alexandria, Egypt.

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

Hutton, G. and Meijer, E. (1988). Monadic parser combinators. *Journal of Functional Programming*, 8:437–444.

Hutton, G. (1992). Higher-order functions for parsing. *Journal of Functional Programming*, 2:323–343, July.

Karttunen, L. and Beesley, K. R. (1992). *Two-Level Rule Compiler. Technical Report ISTL-1992-2*. Xerox Palo Alto Research Center, Palo Alto, CA.

Karttunen, L. (1993). *Finite-state lexicon compiler. Technical Report ISTL-NLTT-1993-04-02*. Xerox Palo Alto Research Center, Palo Alto, CA.

Littell, P., Price, K., and Levin, L. (2014). Morphological parsing of Swahili using crowdsourced lexical resources. In Nicoletta Calzolari, et al., editors, *Proceedings of the Ninth International Conference on Language Resources and Evaluation (LREC'14)*, pages 3333–3339, Reykjavik, Iceland, May. European Language Resources Association (ELRA). ACL Anthology Identifier: L14-1686.

Oard, D. W. (2003). The surprise language exercises. *ACM Transactions on Asian Language Information Processing (TALIP)*, 2(2):79–84, June.

Pan, X., Zhang, B., May, J., Nothman, J., Knight, K., and Ji, H. (2017). Cross-lingual name tagging and linking for 282 languages. In *ACL*.

Pretorius, L. and Bosch, S. E. (2003). Finite-state computational morphology: An analyzer prototype for Zulu. *Machine Translation*, pages 191–212.

Strassel, S. and Tracey, J. (2016). LORELEI language packs: Data, tools, and resources for technology development in low resource languages. In *LREC 2016: 10th Edition of the Language Resources and Evaluation Conference, Portoroz*, pages 3273–3280.

Strassel, S., Bies, A., and Tracey, J. (2017). Situational awareness for low resource languages: the LORELEI situation frame annotation task. In *SMERP2017: First International Workshop on Exploitation of Social Media for Emergency Relief and Preparedness*, Aberdeen.

Wadler, P. (1985). How to replace failure by a list of successes: A method for exception handling, backtracking, and pattern matching in lazy functional languages. In *Conference on Functional Programming Languages and Computer Architecture*, pages 113–128. Springer.