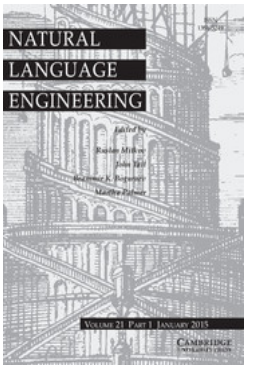
Natural Language Engineering

http://journals.cambridge.org/NLE

Additional services for ***Natural Language Engineering:***

Email alerts: Click here
Subscriptions: Click here
Commercial reprints: Click here
Terms of use : Click here



The Kestrel TTS text normalization system

PETER EBDEN and RICHARD SPROAT

Natural Language Engineering / *FirstView* Article / March 2015, pp 1 - 21
DOI: 10.1017/S1351324914000175, Published online: 12 December 2014

Link to this article: http://journals.cambridge.org/abstract_S1351324914000175

How to cite this article:
PETER EBDEN and RICHARD SPROAT The Kestrel TTS text normalization system. Natural Language Engineering, Available on CJO 2014 doi:10.1017/S1351324914000175

Request Permissions : Click here

CAMBRIDGE JOURNALS

Downloaded from http://journals.cambridge.org/NLE, IP address: 128.218.248.200 on 11 Mar 2015

Natural Language Engineering: page 1 of 21. © Cambridge University Press 2014
doi:10.1017/S1351324914000175

The Kestrel TTS text normalization system

PETER EBDEN[1] and RICHARD SPROAT[2]
[1]*Google, Inc (now at Thought Machine), London, UK*
[2]*Google, Inc, New York, USA*
email: pebden@google.com, rws@google.com

(*Received 28 March 2014; revised 23 October 2014; accepted 27 October 2014*)

Abstract

This paper describes the Kestrel text normalization system, a component of the Google text-to-speech synthesis (TTS) system. At the core of Kestrel are text-normalization grammars that are compiled into libraries of weighted finite-state transducers (WFSTs). While the use of WFSTs for text normalization is itself not new, Kestrel differs from previous systems in its separation of the initial *tokenization and classification* phase of analysis from *verbalization*. Input text is first tokenized and different tokens classified using WFSTs. As part of the classification, detected *semiotic classes* – expressions such as currency amounts, dates, times, measure phases, are parsed into protocol buffers (https://code.google.com/p/protobuf/). The protocol buffers are then verbalized, with possible reordering of the elements, again using WFSTs. This paper describes the architecture of Kestrel, the protocol buffer representations of semiotic classes, and presents some examples of grammars for various languages. We also discuss applications and deployments of Kestrel as part of the Google TTS system, which runs on both server and client side on multiple devices, and is used daily by millions of people in nineteen languages and counting.

1 Introduction

Text-to-speech synthesis (TTS) consists of a number of processing steps that control the conversion of input text into output speech (Taylor 2009). These steps can be broadly broken down into two main categories: linguistic analysis, and synthesis. *Synthesis* involves the actual production of speech, either by selection of units from a database in a unit-selection system, or by generation of parameters in, for example, an HMM synthesizer. Prosody control, including intonation and duration modification, is also part of this phase. *Linguistic analysis* includes text normalization, homograph disambiguation, word pronunciation, and linguistic aspects of prosody prediction such as accent assignment. It is the very first phase, text normalization, that is the topic of this paper.

Text normalization is generally used to refer to such processes as the expansion of abbreviations, the verbalization of digit sequences and the reading of such expressions as times, dates, and currency amounts. Historically this has not been one of the glamorous areas of TTS, and yet it is a critically important part of the process for one simple reason: If the system gets it wrong – if it reads *turn right on 30N* as *turn right on thirty Newtons*, or *$4.5 million* as *four point five dollars million*

– listeners will immediately notice. In that case, it does not matter how good the voice quality is: at best the system will sound like a stupid reader who happens to have a pleasant-sounding voice. At the same time, text normalization is hard: there are times when *30N* should be read as *thirty Newtons*, and while some of these kinds of ambiguities can be handled using sense disambiguation techniques (Yarowsky 1996; Navigli 2009), such techniques typically require at least some annotated data to train models, which is often not available for all of the ambiguities one would like to resolve. As a result, much of the work on text normalization still involves hand-constructed grammars, which are carefully tuned to handle the cases most likely to be encountered. Thus, despite the shift of most of the field of natural language processing towards statistical methods and machine learning solutions, text normalization remains one of the last great bastions of manual techniques.[1]

To start with the obvious, text consists of a sequence of tokens, delimited by punctuation tokens, and (in most languages) spaces. Many of these (non-punctuation) tokens represent words or names, spelled using the normal spelling conventions of the language, as one would find in a dictionary. Following (Sproat *et al.* 2001), we term these *standard words*, or simply *words*. Then there are things like numbers, abbreviations, acronyms, letter sequences, which are not generally found in dictionaries, and are not generally read according to the general pronunciation rules of the language. Again, following (Sproat *et al.* 2001) we call these *non-standard words – NSWs*.[2] Consider the following text:[3]

The Pentagon concluded that Edward Snowden committed the biggest theft of **U.S.** secrets in history, downloading about **1.7** million intelligence files, including information that could put personnel in jeopardy, according to lawmakers.

This text consists mostly of ordinary words and names, but there are four punctuation marks (underlined) and two NSWs (in bold). As is typical of newswire text, this example is fairly 'clean' in that there are relatively few tokens that need to be normalized. On the other hand, for example, ads on the web tend to be much richer in NSWs:

10 lbs. of BodyTech Whey Tech Pro **24** Protein Powder for **$72**.

Similarly, driving directions are often replete with NSWs:

Continue on **NJ-36 S**, Take **NJ-18 N**, County **Rd 537 W**, **NJ-33 W**, **NJ-133 W** and **Co Rd 571**/Princeton Hightstown **Rd** to Blue Jay Way in West Windsor Township.

[1] A reviewer notes that there are other areas where manual techniques are still heavily used, such as gene-name normalization, and even such areas as text classification. To this one could add the point that various components required to make speech technology actually work, such as accurate pronunciation dictionaries, still require substantial manual curation. In any case, it remains true that for the reasons we stated above, text normalization has remained resistant to replacement by machine-learning-based methods.

[2] Also falling into the standard word category are the occasional foreign word or name. These usually do not follow the conventions of the main language, of course, but we classify them with standard words since they are usually ordinarily spelled words in *some* language, and do not naturally fit into the category of NSW.

[3] Source: 'Pentagon Says Snowden Took Most U.S. Secrets Ever: Rogers' *Bloomberg Politics*, January 10, 2014. http://www.bloomberg.com/news/2014-01-10/pentagon-says-snowden-took-most-u-s-secrets-ever-rogers.html

A text normalization system must be able to handle such a range of cases, as well as provide mechanisms for an application to override what the system would do by default. Thus, for driving directions, the calling application *knows* that it is dealing with road names, and thus can pass *NJ-18 N* to the TTS system marked as a road, and expect the TTS system to read it as *New Jersey eighteen north*.

This paper describes the architecture of Kestrel,[4] the text-normalization component of the Google TTS system. Since the main data used by Kestrel are language-specific grammars based on weighted finite-state transducers (WFSTs), we start by briefly describing the history of the use of WFSTs in text normalization.

2 History of FST-based approaches to text normalization

At the core of Kestrel's functionality are grammars, developed in collaboration with native speakers of our target languages, that compile into WFSTs. Recall that WFSTs are machines that have a finite number of *states*, with a designated *initial state*, one or more *final states*, and *directed arcs* connecting states that are labeled with pairs of *input* and *output symbols* taken from an *alphabet* of symbols unioned with the empty symbol ϵ. Arcs may be *weighted* with *costs*, and final states with *exit costs*. These costs may represent probabilities or, more commonly, negative log probabilities; or, as in the present work, they may represent hand-assigned penalties that allow for ranking of different possible analyses.

Finite-state methods for language and speech problems have been with us since the 1950's – see Joshi (1996), and have been widely applied to different problems ranging from early work in morphology (Koskenniemi 1983), parsing (Abney 1996), and machine translation (Bangalore and Riccardi 2001; de Gispert *et al.* 2010). *Weighted* finite-state acceptors and transducers have shown many applications in speech and language processing (Pereira, Riley and Sproat 1994; Mohri, Pereira and Riley 2002; Mohri 2009), and as algorithms have improved, and machines have become more powerful, the range of applications and the sizes of the problems to which they can be applied have concomitantly increased. For large statistical models such as language models, FSTs are typically compiled automatically from tables representing corpus-derived statistics, but in many applications it is desirable to write rules by hand. Compilation algorithms, such as Kaplan and Kay (1994) and Mohri and Sproat (1996), make it possible to write grammars in a human-friendly intuitive way, which are then compiled into WFSTs. Basic details of WFSTs are at this stage widely known, and will not be reviewed here; much discussion can be found in pedagogical natural language processing texts such as Roark and Sproat (2007) and Jurafsky and Martin (2009). In particular, it will be assumed that the reader is familiar with the notion of a *regular relation* between two sets of strings, namely the class of string-set-to-string-set mappings that is computed by an FST. Such relations include many of the string-to-string mappings needed in natural

[4] The various versions of the Google text normalization system have all been named after birds: prior to Kestrel was a system called 'Swift'. At the end of this paper we mention a research system that we call 'Warbler'.

language, but they are limited in that they cannot so readily handle reversals or copying of arbitrary material. In the discussion below, we will see that these latter sorts of phenomena do in fact arise occasionally in text normalization.

Speech synthesizers need to do natural language analysis and as such have tended to make use of methods that are used more widely in the NLP field. Early linguistic analysis systems, such as the DECOMP lexical analysis module of the MITalk speech synthesizer from the 1970s (Allen *et al.* 1987), used a finite-state model of the lexicon, though this was not explicitly modeled using finite-state machines. For text normalization more broadly, MITalk and other early systems tended to use functions written in code.[5]

The first TTS engine that made *systematic* use of FSTs in all aspects of text normalization was the Bell Labs multilingual text-to-speech synthesizer (Sproat 1996, 1997). In that system, the entire analysis of input text, tokenization, normalization, and conversion to phone sequences was accomplished by a cascaded architecture as depicted in Figure 1. The two main components of the text-normalization module were a lexical analysis phase, composed with the input text and converting it into a sequence of words with possible annotations; followed by a grapheme-to-phoneme phase, which converted the words into sequences of phones. In most of the systems developed, the latter was a (composition) cascade of FSTs with hand-built grapheme-to-phoneme rules; see Möbius *et al.* (1997), for an example, for German. The lexical analysis system is more interesting in that it involves:

- A **union** of transducers for individual lexical classes such as numbers, dates, currency amounts, and ordinary words; **concatenated with**
- An FST representing the word separator(s) for the language. For, say, German this would include punctuation and space; for Chinese it would include punctuation, but not space. **Then finally**
- The resulting transducer is converted to the final lexical analysis transducer using transitive closure.

Thus in summary, the lexical analysis system is the union of all of the individual lexical classes, concatenated with the space model, and the result of this is then concatenated with itself zero or more times (transitive closure). Formally, with $\mathscr{L}$ representing the set of lexical classes and $\mathscr{S}$ the space model, this can be expressed as:

$$\left(\bigcup_{l \in \mathscr{L}} l\mathscr{S} \right)^* \tag{1}$$

The transducer thus both tokenizes and lexically analyzes the input simultaneously. Anticipating our discussion below, this is identical to the TokenizeAndClassify phase of Kestrel, but also includes the Verbalize phase, which in Kestrel is a separate component.

[5] For example, the Bell Labs English TTS system from the mid 1980's had a text normalization system called *frend* (for *fr*ont *end*), that was a complex mix of C code and compiled tables.

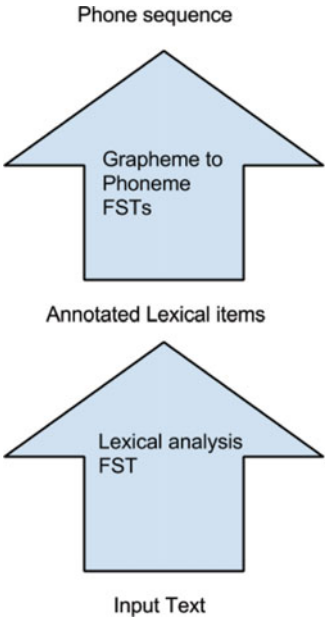


Fig. 1. (Colour online) The text normalization system in the Bell Labs multilingual TTS system (Sproat 1996). Input text is first composed with a lexical analysis transducer, yielding a string of words with possible annotations. These are then passed through a cascade of grapheme to phoneme transducers to yield a phone sequence.

The underlying assumption of such a system that the reading aloud of text can be handled by purely finite-state devices is not trivial, since it makes an implicit claim that the relation between written and spoken language is a regular relation, in the formal sense defined above. More specifically, what we are claiming is largely regular is the relation between what is written in unnormalized text, and the normalized (and ultimately phonetically transcribed) representation from which speech itself is generated. The reason this claim is interesting is that it means that most of the work of the text normalizer can be handled using a single computational device, namely (weighted) finite-state transducers, which we introduced above. Of course, (W)FSTs have been used widely in other areas such as computational phonology and morphology: see, e.g. Johnson (1972), Koskenniemi (1983), Bird and Ellison (1994), and Kaplan and Kay (1994). But to our knowledge, prior to the Bell Labs work nobody had made this claim explicit about the relation between *written and spoken language*. Nonetheless, the assumption works much of the time, because writing systems rarely employ non-regular devices, the exceptions to this being well defined. See Sproat (2000) for a discussion of this topic in the context of a general theory of writing systems. Indeed, given work on using WFSTs as part of the *unit-selection* process in concatenative synthesizers – e.g. Allauzen, Mohri and Riley (2004) – one can treat the entire problem of converting from unnormalized text to the final speech signal as (largely) regular.

One class of exceptions includes currency expressions, which in most languages are *written* with the currency symbol before the number, but *read* with the currency

word after the number expression. Thus '$200' is read as *two hundred dollars* not, obviously, as *dollars two hundred*. It was possible in the Bell Labs system to handle these, but at a cost: every currency symbol would branch a separate set of paths for number expansion. This duplication was the only way that the FST could 'remember' which currency word it should emit at the end. Thus, the transducers for currencies tended to be large and inefficient. Examples like this are one reason why in Kestrel the verbalization phase is handled separately from tokenization and classification.

Following on the Bell Labs work, FST-based approaches to text normalization were pursued at Rhetorical Systems (now part of Nuance) (Skut, Ulrich and Hammervold 2003, 2004), as well as in research projects at the University of Stuttgart (Möbius 2001).

3 Philosophy and design of Kestrel

NSWs frequently exhibit a mismatch between what is written and what is pronounced. This can occur for a variety of reasons. With an abbreviation such as *St* for *Saint*, the mismatch is due to the fact that the word being represented is not spelled in its normal way. For things like numbers and currency amounts, however, the mismatch is because the symbols in question are *ideographic* in the literal sense of that term. Thus *$1.50* represents the currency amount consisting of one dollar and fifty cents. Unlike *St*, which can be said to represent the English word *Saint*, albeit in an abbreviated form, *$1.50* does not represent the English words *one dollar and fifty cents* any more than it represents the Spanish words *un dólar y cincuenta centavos*. Rather it represents the concept directly, and thus can be said to be ideographic.[6] Similarly, a measure phrase such as *3.5 kg* represents the concept of three and a half kilograms, rather than its expression in any particular language. The case with currency is particularly clear since in many languages, the major currency symbol – '$' in the example above – is written before the amount, even though it is spoken *after* the amount.

Furthermore for some types of NSWs, or complex expressions composed thereof, there are several equivalent ways to write the same expression. Thus *Jan 4, 2014* and (in non-US English) *4/1/2014*, both denote and can be read as *January the fourth, twenty fourteen*. These written forms are to a greater or lesser extent divorced from language. This is especially true for constructions like *4/1/2014*, which is really a formal expression for that date that is to a large extent language- and culture-independent. It is this language-independence and 'ideographic' aspect of these expressions that led (Taylor 2009), to term these *semiotic classes*. Kestrel recognizes a large set of these:

- Measure, including percentages
- Currency amounts
- Dates

[6] This correct usage of the term *ideographic* is not to be confused with the incorrect terminology chosen by the Unicode Consortium to refer to the set of Chinese characters.

- Various categories of number:
  - — Cardinal
  - — Ordinal
  - — Decimal
  - — Fraction
- Times
- Telephone numbers
- Electronic addresses

Some of these may include language-particular details, such as *Jan* above as an abbreviation for *January*, something that would work in English or French (*janvier*) but not, for example, in Italian (where the equivalent would be *gen* , for *gennaio*), much less in Russian where it would be янв (for январь). But for the most part these are formal expressions that give little or no indication of how they are to be read.

Various of the semiotic classes listed above can contain other semiotic classes. For example, measure phrases, times, dates, and currency amounts contain fields that may be filled by one of the number classes listed above. These are all defined as embedded messages in a *protocol buffer*[7] definition for each class. Protocol buffers are a representation used throughout Google and beyond to represent data in terms of key-value pairs. The basic unit of a protocol buffer is a *message*, which consists of named keys and typed values, where possible types include integers, strings, booleans or other messages, and where the values may be individual instances or sequences of such types.

Kestrel processes text in two stages. The first is a *tokenization and classification* phase where the text is tokenized into tokens, and each of these is classified into one of four classes:

- An ordinary word
- One of the above semiotic classes
- Punctuation
- Individual, possibly unknown characters (such as ♥ or ♫).

This tokenization and classification phase is handled using FST grammars. The output of the application of these grammars is a textual representation of a linearization – technically, a *serialization* – of a protocol buffer representation of the tokens. This serialization is then parsed into protocol buffers and passed to the second stage, which is *verbalization.*

During the verbalization phase, the protocol buffers are then reserialized, and the semiotic classes tokens are passed to the verbalizer. Protocol buffers have no inherent ordering, and can be serialized in any order. During the verbalization phase, Kestrel actually produces a lattice representing all possible orderings of the components, unless this is specifically overridden by a boolean `preserve_order` field. When the verbalization grammar is applied to this lattice, the orders that match what the

[7] https://code.google.com/p/protobuf/

grammar expects will be verbalized. All others will be ignored. Thus for example, a money token that the tokenizer/classifier parses as

```
money { currency: "usd" amount { integer_part: "50" } ... }
```

(see below), has the currency amount (*usd*) before the number since that is the way it comes in as text (*$50*). During the verbalization phase these are presented in the various permutations:

```
money { currency: "usd" amount { integer_part: "50" } ... }
money { amount { integer_part: "50" } currency: "usd" ... }
...
```

Since English, like most languages, expresses the word for the currency after the number, the verbalization grammar is written to expect input like the second serialization above. Therefore the first ordering will be ignored, and the expression will be verbalized as *fifty dollars*.

Note that such reorderings are problematic for a model that relies exclusively on FSTs to compute the string-to-string mappings as in the Bell Labs architecture (Sproat 1996). As we noted above, while it is possible to handle local reorderings using FSTs, it can be expensive in terms of size and computation speed. Thus, a system that dissociates the surface linear order from the order in which the components are to be verbalized is preferred.

Note also that at the tokenization and classification phase the grammars are only partly language dependent. There is, for example, no need to make language-particular assumptions about the analysis of € *200*. Therefore, a large number of the rules for detecting and parsing semiotic classes are universal, inherited by language-particular tokenization and classification grammars. Verbalization grammars are of course language-dependent.

It will help at this point to work through a specific example. Consider the following text:

```
I need $50k. Please call me at 4:00 at +1-503-444-1234.
```

The serialized protocol buffer representation of the tokenization looks as follows:

```
tokens { name: "I" }
tokens { name: "need" }
tokens { money { currency: "usd" amount { integer_part: "50" }
         quantity: 1000 } }
tokens { name: "." phrase_break: true type: PUNCT }
tokens { name: "Please" }
tokens { name: "call" } tokens { name: "me" }
tokens { name: "at" }
tokens { time { hours: 4 minutes: 0 } }
tokens { name: "at" }
tokens { telephone { country_code: "1" number_part: "503"
                     number_part: "444" number_part: "1234" } }
tokens { name: "." phrase_break: true type: PUNCT }
```

Most of the above is fairly easily interpreted. Thus ordinary words simply have a `name` field that is filled by the spelling of the word. Punctuation fields are similar but also have a `type` field specified as PUNCT as well as indication of whether the punctuation is expected to induce a phrase break.[8]

Currency amounts have fields for the `currency`, the `amount`, which is a number and an optional `quantity`, which corresponds to a portion of the number that is not written as a sequence of digits: in this case the *k* in *$50k* is represented as a `quantity`, rendered during the verbalization phase in English as *thousand* (or just *k*).

Times include fields for hours and minutes, as well as period (AM, PM), and seconds if they are expressed.

Telephone numbers include obvious fields like country codes (if present), extensions (similarly) and various `number_part` components.

The option is also available to the user to provide these fields directly rather than relying on the classifier to make the correct decisions. Thus, the definitions of the various semiotic classes that Kestrel supports are made available via the input API. Since Kestrel parses raw text into the same format, nearly no extra effort is required to accept this additional input form; they are simply passed over by the classifier and sent directly to the verbalizer.

The verbalizer starts by walking over the token stream. It passes over ordinary words and punctuation, and focuses on verbalizing semiotic class instances into sequences of in-lexicon words. This it accomplishes, again, using finite-state grammars. Thus in the above example, the money example, with the currency expression and quantity reordered

```
money { amount { integer_part: "50" } quantity: 1000
        currency: "usd" }
```

is verbalized into *fifty thousand dollars*. The time

```
time { hours: 4 minutes: 0 }
```

becomes *four o'clock*. And the telephone number

```
telephone { country_code: "1" number_part: "503"
            number_part: "444" number_part: "1234" } }
```

is *plus one, five oh three, four four four, one two three four*.

The above sketch presumes a 'pure' interpretation of semiotic classes where their written form has no direct influence on the way they are spoken. While perhaps ideal in principle, such a purist interpretation cannot always work in practice. For example if someone writes (in US English) *2/9/2014*, it is reasonable to read it as any of the following, among others:

[8] A boolean field for phrase breaks is of course insufficient to capture the nuance of all forms of punctuation. A future revision of Kestrel will replace this with a more flexible representation. Note also that phrasing prediction – both determining which punctuation symbols correspond to phrase breaks, and where to place phrase breaks in long unpunctuated word sequences – is outside the scope of Kestrel.

- *two, nine, twenty fourteen*
- *February the ninth, twenty fourteen*
- *the ninth of February, twenty fourteen.*

By converting all of these to a consistent internal representation where the month, day, and year are all represented numerically, we can decide at verbalization time how to express these.

But if someone writes *Feb 9, 2014*, or *February 9, 2014*, while this still *means* the same thing, only the reading *February the ninth, twenty fourteen* is appropriate. That is, while reading *two, nine, twenty fourteen* for the text *Feb 9, 2014* would convey the same information, it would not be faithful to the original text. A user who typed that text, or wanted that text rendered from a web page could legitimately complain if the system produced *two, nine, twenty fourteen* as output. This is especially true in accessibility applications where visually impaired users typically want the system to be as faithful to the input text as possible, while still producing an appropriate-sounding output.[9] The problem is that with the purist interpretation of semiotic classes, inputs like *Feb 9, 2014* also get converted to numerical internal representations for the month, day and year. So how is the verbalizer to remember what the original text was? This is handled in Kestrel in two manners. One is to pass down a `style` field that encodes this information: with the appropriate setting, the system can 'remember' the original text form and verbalize it appropriately. The second way is to relax the requirement that, for example, months be represented internally as numbers, and allow the (possibly normalized) month names to be passed through; in this case the `preserve_order` field is useful to distinguish *9 Feb, 2014* from *Feb 9, 2014*. An earlier version of Kestrel only allowed integers in the protocol buffer `month` field, but that has since been changed to allow for strings, rendering the system less 'pure' but at the same time more flexible in what it can represent.

4 Languages, sample grammar fragments, and technical issues

At the time of writing, Kestrel implementations exist for the following languages:

Cantonese, Danish, Dutch, English, French, German, Hindi, Indonesian, Italian, Japanese, Korean, Mandarin, Polish, Portuguese, Russian, Spanish, Swedish, Turkish, and Thai.

Finite-state grammars for tokenization, classification and verbalization are constructed using an internal toolkit that has also been open-sourced as the Thrax grammar development toolkit; see Tai, Skut and Sproat (2011) and Roark *et al.* (2012) and http://openfst.cs.nyu.edu/twiki/bin/view/GRM/Thrax. Thrax provides a regular-expression syntax, including (transducer) mappings, and context-dependent rewrite rules (Kaplan and Kay 1994; Mohri and Sproat 1996). Samples of Thrax grammars with explanations are given in the ensuing discussion. The reader is referred to the citations given above as well as the OpenFst website for further details of the Thrax system.

[9] T.V. Raman, personal communication.

Many of the grammars, in particular the verbalization grammars, depend on Thrax grammars for number names, grammars that define the ways in which one can read a number name that is written as a sequence of digits. The number name grammars depend in turn on a factorization of the digit sequences into sums of products of powers of ten. Thus, *123* would be factored into $1 \times 10^2 + 2 \times 10^1 + 3$.[10] This could be handled via a Thrax grammar, following Sproat (1996), but since the mechanisms for factorization are largely language-independent, that portion of the conversion is mostly handled in code within Kestrel.[11] Then, a language-specific number verbalization grammar converts these factorized strings into appropriate number names. For highly inflected languages like Russian, where how one reads a number name depends on the context, the grammar produces all possible forms along with features that can be used along with a morphosyntactic tagger to choose the appropriate form; see below for further discussion. At present we have number-name grammars for fifty languages, which are used in both TTS and automatic speech recognition.

As an example, we start by presenting a slightly abbreviated example of fraction verbalization for English. Initially, we define how to read the denominator, which reuses the verbalization rules for ordinal numbers with a small number of special cases. We ensure these special cases are preferred by attaching a small negative weight (lower weights being better in the Tropical semiring) to each:

```
denominator_plural =
    ("1" : "over one" <-1>)
  | ("2" : "halves" <-1>)
  | ("4" : "quarters" <-1>)
  | o.ORDINAL_PLURAL; # Otherwise grab the plural of ordinal words.
```

The rules that read from the marked-up form of the fraction that comes from the classifier reuse the existing cardinal verbalization for the numerator, and our previously defined rule to read the denominator. The rules imported from the *markup* grammar are a set of shared rules that consume the various field names:

```
# e.g. numerator: 2 denominator: 5 -> two fifths
plural = Optimize[
  markup.fraction_numerator
  c.CARDINAL
  markup.fraction_denominator util.ins_space
  denominator_plural
  util.del_space_star
];
```

[10] Non-decimal systems, such as the vigesimal system of Basque, would require some small additional work.

[11] How numbers are factored is dependent on cultural region rather than language. Thus European languages factor based on thousands, East Asian languages on myriads, and so forth. Kestrel handles all of these cases.

We define a similar rule to read fractions where the numerator is *one* and therefore the denominator should be read as a singular form. The main difference is that instead of reading *one third*, we read *a third*, which generally sounds more natural when the fraction is combined with a preceding integer (see below). Note that `a_determiner` is a *word id* that specifies a particular lexical entry in our dictionary, in this case the indefinite article *a*, as opposed to the letter *A*.

```
# e.g. numerator: 1 denominator: 4 -> a quarter
singular = Optimize[
  markup.fraction_numerator
  c.MINUS? ("1" : "a_determiner")
  markup.fraction_denominator util.ins_space
  denominator
  util.del_space_star
];
```

We can also define a composite case for mixed fractions such as *3*. This simply adds a cardinal reading for the integer part, inserts the word *and* and reuses the previously defined rules to read the fraction. Again, a negative weight is applied to prefer the singular reading when applicable.

```
# e.g. integer: 3 numerator: 1 denominator: 2 -> three and a half
composite = Optimize[
  markup.fraction_integer
  c.CARDINAL ("" : " and ")
  ((singular<-1>) | plural)
];
```

We now define several rules to fix up particular cases, for example *an* rather than *a* preceding a vowel.

```
# Converting from a back to one in the simple singular style.
a_to_one = CDRewrite["a_determiner" : "one", "", "", sigma*];
# Correction for "three and a eighth" to "three and an eighth".
a_to_an = CDRewrite["a_determiner" : "an",
                    "", " " util.VOWELS, sigma*];
# Correction to always say "one over one", not "a over one".
fix_over_one = CDRewrite["a_determiner" : "one",
                         "", " over", sigma*];
```

Finally, we define a complete verbalization rule, consuming the outer layer of markup and tying together all previously defined parts. The *export* directive makes the rule available to other grammars which may themselves reuse it; for example, the grammar for measures reuses this rule for constructions such as *three and a half meters*:

```
export FRACTION_MARKUP = Optimize[
  markup.fraction
  util.opening_brace (
```

```
    (singular @ a_to_one) |
    (plural<20>) |
    (composite @ fix_over_one @ a_to_an)
  ) util.closing_brace
];
```

Below, we show a small fragment of a grammar for Russian dates:

```
god = "year" @ nouns;

...

god_masgen = g.F[god, "__MAS,GEN"];

...

year_gen = Optimize[
  (((year_num g.I[" __MAS,GEN"]) @ o_number) g.I[" " god_masgen]) |
   (zero ((d g.I[" __MAS,GEN"]) @ o_number) g.I[" " god_masgen])];

...

dmy_style1_dat = Optimize[
    (m.date_day day_dat m.date_month month_gen
     m.date_year year_gen m.style1)
  @ delfeat @ u.CLEAN_SPACES]
;

...
date_dat = Optimize[
 ( dmy_style1_dat
  | dm_style1_dat
  | my_style1_dat
  | y_style1_dat)
  era?
  g.FeaturesWithCase["DAT"]]
;
```

We start with the verbalization of the word for *year* `год` , which depends on a set of `nouns` defined elsewhere that maps from an English word, into all possible inflected forms of the Russian equivalent. (Note that by default, measures and other terms that are passed from classifier to verbalizer are represented as English words. Thus, *kg* would be mapped to *kilogram* in the classifier. This then gets translated as needed into the particular language of the verbalizer.) Concentrating just on the genitive form of *year* here, the second rule shown calls a function defined in another Thrax grammar that composes the features `__MAS,GEN` with `god` to produce the genitive form. This is then used in an expression for the year which consists

of an ordinal number (`o_number`) in the masculine genitive form, followed by the genitive form of *year*. Finally we can define a day-month-year (DMY) reading in the dative case as involving a dative expression for the day, followed by a genitive [sic] expression of the month, followed by a genitive expression of the year. In the present implementation DMY reading choice is controlled by a 'style' flag – here `style: 1`. Finally, a dative date is defined as the above DMY reading, plus several others, followed by a dative morphosyntactic feature marking. Thus, finally, the dative of *20/10/2000* would be rendered as двадцатому октября двух тысячного года. Deciding whether to use the dative or some other form of the date is outside the scope of these grammars: that is handled by a separate morphosyntactic tagger, a pointwise predictor that uses the grad-boost algorithm to predict each attribute (Duchi and Singer 2009; Neubig, Nakata and Mori 2011).

For some text-normalization applications it is desirable to copy text. An obvious example is morphological reduplication, which may be indicated with a diacritic in text rather than by an actual copy of the material. Thus, Indonesian frequently uses a *2* to represent morphological reduplication: *orang2* for *orang orang*. Such copying is not handled efficiently with ordinary WFSTs. For this reason Kestrel provides a mechanism whereby the verbalizer grammars are checked for the existence of a rule called `REDUP`. If it exists, then the marked-up token to be verbalized is checked for a match against that rule. If it matches, then a copy is made in code (a concatenation of the FST representing the input, with itself), and the duplicated string is added to the input lattice. The verbalization then proceeds with both the original input, and the reduplicated input.

This mechanism, necessary in any case for phenomena such as morphological reduplication, is also convenient for dealing with other cases where a feature is essentially copied. For example, consider a marked-up currency amount such as:

```
money { amount { integer_part: 1 fractional_part: 20 }
        currency: "usd" }
```

We want to read this as *one dollar and twenty cents*, so we want the major currency word *dollar* to go with the `integer_part` of the currency, and the appropriate minor currency term *cents* to go with the `fractional_part`. It would be most convenient, therefore if the input were something like:

```
money { amount { integer_part: 1 } currency: "usd_maj" }
money { amount { fractional_part: 20 } currency: "usd_min" }
```

This can easily be achieved using the `REDUP` mechanism, first by copying the whole marked up expression, and then operating on the left and right copy independently to yield the final verbalization. Note that to do this using purely regular relations would necessitate a copy rule for every currency amount. This is a large number – ISO4217 defines around 150 currencies – and for each minor the WFST would have to remember the associated major currency, resulting in a lot of duplication of the intermediate structure, and an explosion in the size of the FST.

We note in passing that for languages such as Chinese, Japanese, and Thai, word segmentation is needed as a preprocessing step to the Kestrel grammars. At the

time of writing, our Chinese system uses the Google-internal CJK segmenter, the Japanese system uses an adapted version of Mecab (http://mecab.sourceforge.jp), and the Thai system uses a conditional random field-based segmenter.

5 Application domains and deployment info

Kestrel has been designed to be a universal solution to all the text normalization needs of Google's text-to-speech system. Hence, it needs to handle inputs in many forms. The most obvious of these is raw text, which comes from many sources such as when the text-to-speech system is used to read web pages, books, e-mails or calendar appointments. In such cases, no hints are available as to how to normalize any NSWs encountered, and a correct reading may require dealing with many of them. For example:

- The *100–200* seat narrow-body – or single-aisle – aircraft market is forecast to generate *$20 trillion* (*12.8tn*) over the next *20* years.
- Skyfall (*2012*), the new James Bond *007* movie.
- *4pm pebden:rws 1:1.* Attending?

Some of the inputs Kestrel encounters may be partly pre-normalized, such as with driving directions. For example, the directions shown on screen might be (the somewhat fictitious) *Turn right onto AK-47 N*, but when known to be safe, the text sent to Kestrel may be expanded to *Turn right onto Alaska forty seven north.* In general, such domain knowledge is beyond the scope of Kestrel; obviously, in most cases the *AK* in *AK-47* would *not* expand to *Alaska.*

As we noted above, Kestrel also supports receiving fully structured inputs. The most common usage of this at present is for answers to spoken queries; e.g. the question *What is ten US dollars in Malaysian ringgit* would generate an answer like *10 US dollars equals 33.08 Malaysian ringgit*. In this case, the entities in the spoken answer are well known, and so the input to Kestrel is presented as:

```
money { currency: "usd" amount { integer_part: "10" } }
text: "equals"
money { currency: "myr"
        amount { integer_part: "33" fractional_part: "08" } }
```

This removes any potential ambiguity in the input text, and Kestrel is hence able to skip classification for most of the input and proceed directly to the verbalization phase.

Kestrel exists in several deployments. Perhaps the most publicly visible example is as part of the Google text-to-speech app for Android (https://play.google.com/store/apps/details?id=com.google.android.tts), which is currently installed on several hundred million Android devices. Deployment to many remote, portable devices presents some specific challenges; notably, the most compact voice packs are downloaded on demand and have total size around 4MB. The text-normalization data is of course only part of the voice data, and so Kestrel is required to fit into approximately 800kb in this incarnation. This is achieved partly by aggressive

compaction of the FSTs shipped with the Kestrel implementation. This reduces the size of the deployed FSTs by approximately an order of magnitude versus the standard *VectorFst* serialization. (See http://www.openfst.org for details of the different FST representations.) At the same time, close attention is paid to what we add to the Kestrel grammars and how it is written in order to avoid increasing their size unnecessarily.

The Android deployment also places requirements on Kestrel with regards to performance. Since it must run before synthesis takes place, and one of the common use cases for the local synthesizer is as a screen reader whose users are very sensitive to latency, its performance is a significant concern. This is exacerbated by running on cellphones, which still have significantly less computing power than is available on a server-based implementation.

The significant majority of computation time in Kestrel goes into the *tokenization and classification* phase. A major reason for this is that it is highly non-deterministic, since many parts of the output markup must be emitted before consuming the input that determines whether they are valid. This is significantly mitigated by integrating a *look-ahead filter* (Allauzen, Riley and Schalkwyk 2011) which prunes many invalid intermediate states that will not lead to connected output states.

Further gains are made simply by care in writing the grammars; for example, disallowing the tokenization of, say, 'hello' into two tokens 'hel' and 'lo'. The single word analysis would win in any case, but allowing the possibility of splitting it into two generates unwanted states and arcs which in turn consumes valuable additional CPU time.

Kestrel also ships as an embedded system on other Android devices, or via downloadable high-quality voice packs for Android. These have relaxed size requirements versus the compact Android voices, but similar considerations about performance apply. Finally, Kestrel also runs on Google's servers as part of the TTS implementation used to answer, via voice, millions of queries per day. This deployment has the loosest limits on file size and the greatest computational power available.

6 Evaluation

Measuring the efficacy of a text normalization system is of course dependent on having an appropriate data set to evaluate it against. There is, unfortunately, no appropriate publicly available data that we can use to evaluate our system. There are datasets for text normalization of social media such as Twitter (e.g. the Edinburgh Twitter Corpus), but Kestrel's current text-normalization grammars are not designed to handle such data.

We do have test sets that were developed for internal use at Google and represent the kinds of material that our synthesizers are currently most called upon to handle, such as driving directions, and factoid answers to user queries. Unfortunately we cannot release the contents of these test sets, but we can report on the performance. This we do in Tables 1 and 2. Table 1 gives results for English, a relatively easy language, broken down by semiotic class. For each class we give the number of

Table 1. *Kestrel's accuracy on English*

Cardinal	1000/1000 (100%)
Date	1116/1116 (100%)
Decimal	1000/1000 (100%)
Driving directions	656/1000 (65.6%)
Electronic	430/430 (100%)
Fraction	673/673 (100%)
Measure	1415/1559 (90.8%)
Money	568/568 (100%)
Ordinal	1000/1000 (100%)
Roman numeral	1287/1483 (86.8%)
Telephone	948/1008 (94.1%)
Time	715/1000 (71.5%)
Bug	508/541 (93.9%)

Table 2. *Kestrel's accuracy on Russian. Some categories require explanation. 'Connector' denotes to the reading of '–' between numbers, dates, etc., which may be read in various ways depending upon the intended meaning. 'Letter sequence' denotes the reading of 'acronyms' as sequences of letters versus, for example, as a word. 'Morphosyntactic homographs' are homographs that can be disambiguated on the basis of morphosyntactic features, such as number and case. 'Transliteration' involves cases that need to be transliterated from Roman into Cyrillic script in order to be pronounced in Russian.*

Address	15/15 (100%)
Cardinal	74/74 (100%)
Connector	21/21 (100%)
Date	142/142 (100%)
Decimal	125/158 (79.1%)
Digit	194/194 (100%)
Electronic	213/213 (100%)
Fraction	88/88 (100%)
Letter sequence	17/17 (100%)
Measure	330/340 (97.1%)
Money	138/225 (61.3%)
Morphosyntax homographs	144/144 (100%)
Ordinal	53/53 (100%)
Roman numeral	14/14 (100%)
Telephone	12/12 (100%)
Time	100/100 (100%)
Transliteration	3/3 (100%)
Verbatim	134/134 (100%)
Bug	140/155 (90.3%)

examples in the test, and the number and percentages that were correctly handled. The 'bug' set denotes specific text-normalization bugs of a variety of types that have been reported by users. The percentages reported there reflect the proportion of cases that have since been fixed. Of particular note are the 'driving directions' set, which are particularly tricky since many require expansions of ambiguous abbreviations

(such as *N* for *North*, or *Staten Is* for *Staten Island*). Overall, Kestrel achieves **91.3%** correct for English.

Table 2 reports similar results for Russian. Since Russian is a younger system, we have a smaller set of evaluation data. In particular, the trickier test cases such as driving directions are not yet available, but on the tests we do have, Kestrel achieves **93.1%** correct. Note also that mistakes in Russian often derive from errors in the statistical morphosyntactic tagger, discussed above, which is strictly speaking external to Kestrel.

One obvious question that also arises is how long it takes to develop a system for a new language. Clearly this depends on the difficulty of the language: Russian and several other Slavic languages require *significantly* more time to develop than, say, English, due to the complexity of the morphology. For an easy language, it is possible to put together an initial system with a couple of weeks' worth of work, though of course there will be a substantial amount of development and maintenance required after that initial construction. For a language like Russian, a few months' worth of work is required, though in our experience the development of other Slavic languages, such as Polish, has gone a lot more quickly since the grammars can often be adapted from those for Russian.

7 Future work

A number of pieces of work are ongoing that will enhance Kestrel.

Thrax includes the ability to specify *weighted pushdown transducers* (Allauzen and Riley 2012), and indeed this mechanism is already used in Kestrel grammars for some languages. It is being extended to *weighted multistack pushdown transducers*, that involve more than one stack (Aho 1969), and which are capable of performing copy operations. This will provide a more principled way to handle phenomena currently handled by the `REDUP` mechanism described above.

There is, naturally, an active interest in learning models of text normalization from unannotated or lightly annotated data. For one relatively recent example, particularly relevant to the functionality of Kestrel, see Sproat (2010). While we expect that many components of the text-normalization process are likely to require hand-developed grammars for the foreseeable future, we are pursuing research to try to replace portions of the system with automatically learned systems. Recent work reported in Roark and Sproat (2014), for example, supplements our English system's (hand-built) abbreviation expansion system with abbreviation expanders learned semiautomatically from data. Crucially different from previous work along these lines, though, the system reported in Roark and Sproat (2014) aims to 'do no harm', by only expanding abbreviations that the system is very confident about given the context – essentially trading recall for very high precision.

An intriguing possibility would be to allow the classification step to emit multiple hypotheses, which a trained system could then choose between. This would provide a cleaner separation between the classification and tokenization step, which is readily expressed in FSTs, and the decision between ambiguous options, which often needs

to be informed by other sentence features which are not practical to express in an FST.

Finally, we discussed at the beginning of the paper the purist notion whereby the identification of semiotic classes is completely separate from their verbalization. As we noted there, this purist notion is not tenable in general since many expressions that must be normalized, nonetheless may give some clues as to how they should be verbalized in their written representation. Indeed, when one pursues this line of reasoning further one sees that the verbalization of semiotic classes gives clues as to how they may sometimes be identified in text. For example, if we know that a date may be verbalized as *January the third, two thousand six*, then one might expect that this could be found in text as *Jan 3, 2006*, *January the 3rd, 2006*, and so forth. Furthermore, when written in these ways, as we noted above, we would expect them to be *read* as *January the third, two thousand six* and not (in US English) as *one, three, two thousand six*. It seems redundant to have separate classifier rules that recognize these expressions, and in such cases there should be a tight link between what is written and how it is verbalized. Kestrel does not really enforce this, except via tricks such as the `style` flags as noted above. We are working on an experimental system, called Warbler, that allows a closer integration of classification and verbalization, where that is appropriate.

Acknowledgments

We would like to acknowledge the contributions of the following people to the development of the Kestrel text normalization system.

First and foremost, Paul Taylor, who conceived of and developed the first version of the Kestrel system during his time at Google.

Second, to Alexander Gutkin, for his work on FST serialization, grammars, and general Kestrel development.

Third, the many individual contributors of grammars and portions of grammars used for various languages. To date: Alex Balabanov, Asmund Bekker, Sofia Bernardo, Benoît Brard, Sibel Ciddi, Daniel van Esch, Niels Egberts, Carlo Di Ferrante, Chanwoo Kim, Zu Kim, Deo Liang, Viviana Montoya, Tasuku Oonishi, Knot Pipisriswat, Sergio Sancho, Andrey Shulyatyev, Fabian Siddiqi, Mateusz Westa, Heiga Zen, and Zoey Zhang.

We also thank Ian Hodson for his management and support of the text-to-speech group at Google.

Finally, we acknowledge the comments of three anonymous reviewers for *Natural Language Engineering*.

References

Abney, S. 1996. Partial parsing via finite-state cascades. *Natural Language Engineering* **2**(4): 337–344.

Aho, A. 1969. Nested stack automata. *Journal of the Association for Computing Machinery* **16**(3): 383–406.

Allauzen, C., Mohri, M., and Riley, M. 2004. Statistical modeling for unit selection in speech synthesis. In *Proceedings of the 42nd Annual Meeting of the Association for Computational Linguistics (ACL'2004)*, pp. 55–62.

Allauzen, C., and Riley, M. 2012. A pushdown transducer extension for the OpenFst library. In *Conference on Implementation and Application of Automata*, Lecture Notes in Computer Science vol. 7381, Heidelberg: Springer, pp. 66–77.

Allauzen, C., Riley, M., and Schalkwyk, J. 2011. Filters for efficient composition of weighted finite-state transducers. In *Conference on Implementation and Application of Automata*, Lecture Notes in Computer Science vol. 6482, Heidelberg: Springer, pp. 28–38.

Allen, J., Hunnicutt, M. S., Klatt, D., Armstrong, R., and Pisoni, D. 1987. *From Text to Speech: The MITalk System*, Cambridge, England, UK: Cambridge University Press.

Bangalore, S., and Riccardi, G. 2001. A finite-state approach to machine translation. In *2nd Meeting of the North American Chapter of the Association for Computational Linguistics*, Pittsburgh, PA, pp. 1–8.

Bird, S., and Ellison, T. M. 1994. One-level phonology: autosegmental representations and rules as finite automata. *Computational Linguistics* **20**(1): 55–90.

de Gispert, A., Iglesias, G., Blackwood, G., Banga, E., and Byrne, W. 2010. Hierarchical phrase-based translation with weighted finite-state transducers and shallow-*n* grammars. *Computational Linguistics* **36**(3): 505–533.

Duchi, J., and Singer, Y. 2009. Boosting with structural sparsity. In *Proceedings of the 26th International Conference on Machine Learning*, Montreal, p. 297304.

Johnson, C. D. 1972. *Formal Aspects of Phonological Description*. Walter de Gruyter.

Joshi, A. 1996. A parser from antiquity. *Natural Language Engineering* **2**(4): 291–294.

Jurafsky, D., and Martin, J. 2009. *Speech and Language Processing: an Introduction to Natural Language Processing, Computational Linguistics, and speech recognition*. 2nd edn. Pearson: Prentice Hall.

Kaplan, R. M., and Kay, M. 1994. Regular models of phonological rule systems. *Computational Linguistics* **20**: 331–378.

Koskenniemi, K. 1983. Two-level morphology: a general computational model of word-form recognition and production. *PhD thesis*, University of Helsinki.

Möbius, B. 2001. *German and Multilingual Speech Synthesis*. Phonetik AIMS: Arbeitspapiere des Instituts für Maschinelle Sprachverarbeitung vol. 7, Lehrstuhl für experimentelle Phonetik, Stuttgart.

Möbius, B., Sproat, R., van Santen, J., and Olive, J. 1997. The Bell Labs German text-to-speech system: an overview. In Eurospeech. Rhodes.

Mohri, M. 2009. Weighted automata algorithms. In M. Droste, W. Kuich, and H. Vogler (eds.) *Handbook of Weighted Automata*, Monographs in Theoretical Computer Science, Springer, pp. 213–254.

Mohri, M., Pereira, F. C. N., and Riley, M. 2002. Weighted finite-state transducers in speech recognition. *Computer Speech and Language* **16**(1): 69–88.

Mohri, M., and Sproat, R. 1996. An efficient compiler for weighted rewrite rules. In *Proceedings of the 34th Annual Meeting of the Association for Computational Linguistics*, pp. 231–238.

Navigli, R. 2009. Word sense disambiguation: a survey. *ACM Computing Surveys* **41**(2): 169.

Neubig, G., Nakata, Y., and Mori, S. 2011. Pointwise prediction for robust, adaptable Japanese morphological analysis. In *Association for Computational Linguistics*, Portland, OR, pp. 529–533.

Pereira, F., Riley, M., and Sproat, R. 1994. Weighted rational transductions and their application to human language processing. In *ARPA Workshop on Human Language Technology*, Plainsboro, NJ, pp. 249–254.

Roark, B., Riley, M., Allauzen, C., Tai, T., and Sproat, R. 2012. The OpenGrm open-source finite-state grammar software libraries. In*ACL*, Jeju Island, Korea, pp. 61–66.

Roark, B., and Sproat, R. 2007. *Computational Approaches to Morphology and Syntax*. Oxford: Oxford University Press.

Roark, B., and Sproat, R. 2014. Hippocratic abbreviation expansion. In *Association for Computational Linguistics*, Baltimore, MD, pp. 364–369.

Skut, W., Ulrich, S., and Hammervold, K. 2003. A generic finite state compiler for tagging rules. *Machine Translation* **18**(3): 239–250.

Skut, W., Ulrich, S., and Hammervold, K. 2004. A bimachine compiler for ranked tagging rules. In *Proceedings of the 20th International Conference on Computational Linguistics, COLING '04*, Association for Computational Linguistics, Geneva, Switzerland, pp. 198–204.

Sproat, R. 1996. Multilingual text analysis for text-to-speech synthesis. *Natural Language Engineering***2**(4): 369–380.

Sproat, R. (ed.): 1997. *Multilingual Text-to-Speech Synthesis: The Bell Labs Approach*. Boston, MA: Springer.

Sproat, R. 2000. *A Computational Theory of Writing Systems*. Cambridge, England, UK: Cambridge University Press.

Sproat, R. 2010. Lightly supervised learning of text normalization: Russian number names. In*IEEE Workshop on Spoken Language Technology*, IEEE, Berkeley, CA, pp. 436–441.

Sproat, R., Black, A., Chen, S., Kumar, S., Ostendorf, M., and Richards, C. 2001. Normalization of non-standard words. *Computer Speech and Language* **15**(3): 287–333.

Tai, T., Skut, W., and Sproat, R. 2011. Thrax: an open source grammar compiler built on OpenFst. In *Automatic Speech Recognition and Understanding Workshop*, Waikoloa Resort, Hawaii.

Taylor, P. 2009. *Text to Speech Synthesis*. Cambridge, England, UK: Cambridge University Press.

Yarowsky, D. 1996. Homograph disambiguation in text-to-speech synthesis. In J. van Santen, R. Sproat, J. Olive, and J. Hirschberg (eds.), *Progress in Speech Synthesis*, New York: Springer, pp. 157–172.