VoiceData Text Normalization

Automatic Speech Recognition (ASR) Data Generator Toolkit

Preface By Richard Sproat
Research & Machine Intelligence,
Google, New York

Author: Zoltán Somogyi

AI-TOOLKIT: https://ai-toolkit.blogspot.com

TOOLKIT AI ARTIFICIAL INTELLIGENCE TOOLKIT
THE APPLICATION OF AI MADE EASY!

Preface By Richard Sproat Research Scientist at Google

Finite-state methods have had a long history in computational linguistics, dating from early work on parsing by Aravind Joshi and others at the University of Pennsylvania in the 1950's. By the early 1980's computers were powerful enough that it was possible to develop wide-coverage finite-state solutions to some problems, such as Kimmo Koskenniemi's (1983) morphological analyzer for Finnish. Koskenniemi's solution involved building the relevant FSTs by hand, but around that time, Ron Kaplan and Martin Kay developed their algorithm (only published in 1994) for compiling rewrite rules of the form:

$$\varphi \rightarrow \psi / \lambda \_ \rho$$

into FSTs. Rules of this form had been the main workhorse of phonology through the 1970's and their instantiation as FSTs meant that one could build linguistic descriptions of phonological or morphological systems that could then be compiled down into a computational model.

Later (in 1996), Mehryar Mohri and I made some algorithmic improvements to the Kaplan and Kay system and extended it to compile *weighted* rules into *Weighted* Finite State Transducers. Around that same time I and others at Bell Laboratories did the first work on applying WFSTs to the problem of text normalization. With the development of ever faster processors, increased memory, and improvements to algorithms, practical WFST-based solutions have become mainstream in many applications including language modeling, automatic speech recognition and document entity extraction.

The work that we started over 20 years ago at Bell Laboratories is still used in a modern incarnation as part of the Google text-to-speech system, and though we have recently started investigating neural methods, the deployed systems still have finite-state grammars as the core machinery used in text normalization. Text-normalization grammars are written in Thrax (open-sourced at http://www.openfst.org/twiki/bin/view/GRM/Thrax), and the compiled WFSTs in the OpenFst format (http://www.openfst.org/) are used at runtime by the Kestrel text-normalization system, a pared-down version of which is open-sourced as Sparrowhawk (https://github.com/google/sparrowhawk). One great advantage of this approach is it allows a clean separation of algorithms and run-time code from language-particular data: one merely needs to provide a linguistic description that can be compiled down into WFSTs, and the run-time algorithm works in a language-independent fashion.

While we have made good use of these packages, especially OpenFst and Thrax, within Google, our reason for open-sourcing them was of course to get a much wider adoption of what we hoped would be useful tools for the wider community. It is therefore gratifying that Zoltán Somogyi has made significant use of our software in the Text Normalization module for *VoiceData*, the *Automatic Speech Recognition Data Generator Toolkit*. I hope that his work will benefit many.

Richard Sproat

Research & Machine Intelligence
Google, New York

Acknowledgments

This work and the accompanied software would not have been possible without the huge amount of work of several people at Google Inc. as Richard Sproat, Terry Tai, Wojtek Skut, Cyril Allauzen, Kyle Gorman, Michael Riley, Martin Jansche, Peter Ebden, etc.

Richard Sproat was a great help during making of the different grammar features and therefore special thanks to him personally.

For further acknowledgments of the accompanied software please consult the software documentation.

Table Of Contents

.

Introduction

The first step in data generation for Automatic Speech Recognition (ASR) is the normalization of the input text. Normalization means that all non text elements (e.g. 1/1/2018 or 10.5kg) must be converted to simple text (as we speak it). This is a complex task because the normalization must use language dependent grammar rules in order to be able to detect and normalize such elements automatically.

VoiceData text normalization is based on Sparrowhawk/Kestrel, the Google-internal TTS text normalization system reported in Ebden and Sproat (2014).

This book will explain how to use the VoiceData text normalization system also including the detailed explanation of how to develop the language dependent grammars. The grammars are expressed as regular expressions and context-dependent rewrite rules which are then compiled into weighted finite-state transducers in the OpenFst format. These FST's are then stored in far archives (files with 'far' extension). These archives are then used during the normalization process which consists of three steps, first the segmentation to sentences, then a tokenization/classification step (the detection of the items needed to be normalized and separation from the rest of the text) and then a verbalization step (the verbalization of the items needed to be normalized). The result is a normalized text in which all of the in the grammars defined items (including the so called semiotic classes) are verbalized according to the definitions in the grammars.

You will be able develop better grammars if you know more about finite-state transducers (FST's) and about how to use them for natural language processing. For this reason Chapter 4 contains an extensive introduction to Finite State Transducers. Modeling natural language features is a very complex subject. The aim of this chapter is to explain in simple lay terms why we need Finite State Transducers (FST's), how they work and how they relate to the text normalization process.

Chapters 2, 3 and 5 can also be used as a reference for grammar functions, symbols and semiotic classes in the normalization system.

Chapter 6 explains the most important things about the VoiceData user interface. For more information please consult the built-in help in the right sidebar of the software.

Chapter 1 The Basics

The Basics Of Normalization

Introduction

The first step in data generation for Automatic Speech Recognition (ASR) is the normalization of the input text. Normalization means that all non text elements (e.g. 1/1/2018 or 10.5kg) must be converted to simple text (as we speak it). This is a complex task because the normalization must use language dependent grammar rules in order to be able to detect and normalize such elements automatically.

This chapter will explain how to use the VoiceData text normalization system also including the detailed explanation of how to develop the language dependent grammars. The grammars are expressed as regular expressions and context-dependent rewrite rules which are then compiled into weighted finite-state transducers in the OpenFst format. These FST's are then stored in far archives (files with 'far' extension). These archives are then used during the normalization process which consists of three steps, first the segmentation to sentences, then a tokenization/classification step (the detection of the items needed to be normalized and separation from the rest of the text) and then a verbalization step (the verbalization of the items needed to be normalized). The result is a normalized text in which all of the in the grammars defined items (including the so called semiotic classes) are verbalized according to the definitions in the grammars.

You will be able develop better grammars if you know more about finite-state transducers and about how to use them for natural language processing, but you should understand most of the subjects in this chapter without knowing much about FST's.

Grammar Data Files

In VoiceData grammar definitions are stored in the 'normdata' directory under the program folder. Each language has its subdirectory designated with the ISO two letter language code (e.g. 'en' is English, 'nl' is Dutch). The root directory per language contains the following required configuration files:

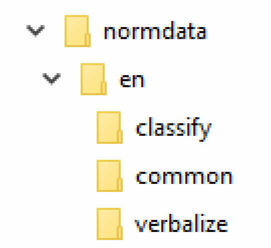
- norm_config.tnm - contains the normalization (classification and verbalization) settings for the specific language.
- grammar_config.tnm - contains the list of root grammar definitions to compile. Grammar dependences will be detected automatically.

> **The en_ex example grammar**
>
> After you install VoiceData you will notice a subdirectory called “en_ex” under the “normdata” directory normally located in:
>
> C:\Program Files\VOICEDATA\normdata\en_ex
>
> This is an example English grammar containing all grammar source codes (grm files). In order to be able to use this grammar in VoiceData you will have to rename the current “en” folder to for example “en_final” and the “en_ex” folder to “en”. In this way you will be able to use the example grammar as the English grammar in VoiceData. Make sure that you do not delete the full English grammar but rename it that you can restore it later! Please note that the full English grammar does not contain the source grm files!
>
> The “en_ex” grammar contains a simplified grammar which can be used for experimentation. It also contains all of the examples explained in this book.

Simplified English Grammar Example

As an example we will go through a simplified grammar definition for English hereunder.

The directory structure looks like this:



```
normdata
  en
    classify
    common
    verbalize
```

The ‘en’ folder is the root directory for English. The ‘classify’ folder contains the grammar definitions for the classification step. The ‘verbalize’ folder contains the grammar definitions for the verbalization step. The folder ‘common’ contains grammar definitions used in both classification and verbalization.

The ‘en’ directory contains the following files:

- norm_config.tnm
- grammar_config.tnm
- tokenizer.tnm
- verbalizer.tnm
- verbalizer_serialization_spec.tnm
- sentence_boundary_exceptions.tnm

The norm_config.tnm configuration file in the ‘en’ folder contains the following text:

```
tokenizer_grammar:  "normdata\\en\\tokenizer.tnm"

verbalizer_grammar:  "normdata\\en\\verbalizer.tnm"

sentence_boundary_regexp: "[\\.:!\\?] "

sentence_boundary_exceptions_file:
"normdata\\en\\sentence_boundary_exceptions.tnm"

serialization_spec: "normdata\\en\\verbalizer_serialization_spec.tnm"
```

Please note that for all file references relative paths are used as for example "normdata\\en\\tokenizer.tnm". This is because the normdata directory is just a subdirectory under the program folder (where the VoiceData application is). Note also that the directory separator is a double backslash ‘\\’.

The first line defines the configuration file for the classification (tokenize) step. The second for the verbalization step. The third line defines a regular expression for the detection of sentence boundaries, the end of the sentences (obviously a dot, a question mark, etc.). The fourth line defines the configuration file where sentence boundary exceptions are defined. Sentence boundary exceptions are e.g. ‘Mr.’ where the dot does not mean the end of the sentence but an abbreviation, etc. In order to prevent the false detection of the end of sentences add to this file all possible sentence boundary exceptions (see later).

tokenizer.tnm contains the following text:

```
grammar_file: "normdata\\en\\classify\\tokenize_and_classify.far"

grammar_name: "TokenizerClassifier"

rules { main: "TOKENIZE_AND_CLASSIFY" }
```

Where tokenize_and_classify.far defines the compiled far archive used in the classification step. The main rule, which is exported from the far archive, is defined by the last line and called ‘TOKENIZE_AND_CLASSIFY’. We will see this later in more detail when talking about the details of the grammars.

verbalizer.tnm contains the following text:

```
grammar_file: "normdata\\en\\verbalize\\verbalize.far"

grammar_name: "Verbalizer"

rules { main: "ALL" }
```

Where verbalize.far defines the compiled far archive used in the verbalization step. The main rule, which is exported from the far archive, is defined by the last line and called ‘ALL’. We will see this later in more detail when talking about the details of the grammars.

sentence_boundary_exceptions.tnm contains the following text:

```
Mr.

Dr.
```

```
Mrs.
St.
…
```

This is just a per line list of all exceptions, which should not be wrongly identified as sentence endings.

The Serialization Specification

> Please note that several grammar specific things in this section will be explained later, therefore if you do not understand everything then come back to this section later!

The line

‘serialization_spec:"normdata\\en\\verbalizer_serialization_spec.tnm" ‘

in the configuration file specifies the serialization specification file. The serialization specification should contain a definition for all semiotic classes which you want to use in your grammars. These definitions determine how the semiotic classes will be handled in the verbalization step. It is called serialization because the tokenizer serializes (stores) the grammar rules and sends them to the verbalizer in the format defined in the serialization specification file.

For the money semiotic class we define the following in our English grammar:

```
class_spec {
  semiotic_class: "money"
  style_spec {
    record_spec {
      field_path: "money.amount.integer_part"
      suffix_spec {
        field_path: "money.currency"
      }
    }
    record_spec {
      field_path: "money.amount.fractional_part"
      suffix_spec {
        field_path: "money.currency"
      }
    }
  }
}
```

The following sections will explain the above in detail.

Class Specification Section

This section is the specification of a serialization format for a particular semiotic class [2].

Designation	`class_spec {}`
Fields	`semiotic_class:` The name of the semiotic class.
	`style_spec {}` "Denotes the style within the semiotic class." Enabling multiple ways (styles) of verbalizing the same semiotic class [2].

Style Specification Section

This section is the specification for serializing a semiotic class in a particular style [2].

Designation	`style_spec {}`
Fields	`record_spec {}` "Gives the specification for how tokens should be serialized in this style. The serialization components for this style will be emitted in the same order as the record specs in this field" [2].
	`required_fields:, prohibited_fields:` "A single instance can have multiple fields separated by "|".

Record Specification Section

This section is the specification for serializing a sub-part of a semiotic class. Record Specifications may be simple, such as a single field, or recursively combine additional record specifications to specify more elaborate formats. For a repeated scalar field, we simply serialize all the values in the token for this field in an identical fashion, respecting the original order [2].

Designation	`record_spec {}`
Fields	`prefix_spec {}` "The serialization for these Record Specifications will be emitted prior to every instance of the main field for this spec" [2].
	`suffix_spec {}` "The serialization for these Record Specifications will be emitted after every instance of the main field for this spec" [2].

	`field_path:`
	Defines the path from the top-level token to the field handled by this Record Specification in the output from the tokenization/classification. “If the label field is not set, the terminal portion of this will be used as the label in the serialized output” [2]. For example if the output from the tokenization is
	`money { currency: "usd" amount { integer_part: "3" fractional_part: "50" } }`
	then the path will be 'money.amount.integer_part' or 'money.amount.fractional_part'. The label will be 'integer_part' or 'fractional_part' respectively.

	`label:` “Defines the record label in the serialization. This should be set only to override the use of the terminal field name from the field path as the default label” [2].
	`default_value:` “String defining the value to be used for the field in case it is not set” [2].

The specification for the date class can be the following:

```
class_spec {
  semiotic_class: "date"
  style_spec {
    record_spec {
      field_path: "date.day"
    }
    record_spec {
      field_path: "date.month"
    }
    record_spec {
      field_path: "date.year"
    }
  }
  style_spec {
    record_spec {
      field_path: "date.month"
    }
    record_spec {
      field_path: "date.day"
    }
    record_spec {
      field_path: "date.year"
    }
  }
}
```

> Note in the above example the two separate Style Specification sections which define the two date styles (DMY and MDY). This is important if you want to be able to normalize these two different date formats! The order of the fields will be kept.

The specification for a simple cardinal number class:

```
class_spec {
  semiotic_class: "cardinal"
  style_spec {
    record_spec {
      field_path: "cardinal.integer"
    }
  }
}
```

> There is a separate chapter (Chapter 3) about semiotic classes with all of the classes explained together with their serialization specifications!

The grammar_config.tnm configuration file in the 'en' folder contains the following text:

```
normdata\\en\\classify\\tokenize_and_classify.grm
normdata\\en\\verbalize\\verbalize.grm
```

These two grammar files are the main or root grammar files. The tokenize_and_classify.far and verbalize.far archives are compiled from these two grammars. We use the name root grammar because most of the time these grammar files depend on several other grammar files, as we will see later. VoiceData automatically determines all dependences (other grammar files linked to the root grammar) and will compile all grammar files according to the required sequence of linkage.

As an example let us see now what is in the verbalize.grm file:

```
import 'normdata\en\verbalize\date.grm' as d;
import 'normdata\en\verbalize\measure.grm' as M;
import 'normdata\en\verbalize\money.grm' as m;
import 'normdata\en\verbalize\numbers.grm' as n;
import 'normdata\en\verbalize\time.grm' as t;
import 'normdata\en\verbalize\verbatim.grm' as v;
# Combines all of the semiotic classes together.
export ALL = Optimize[d.DATE | M.MEASURE | m.MONEY | n.CARDINAL_MARKUP |
t.TIME | v.VERBATIM];
```

The grammar definition language will be explained later but some elements may be interesting at this stage already. The lines starting with the 'import' statement define the grammar files containing grammar definitions used by verbalize.grm. For example how to handle calendar dates is defined in the grammar file date.grm. The 'as d' statement just assigns a name to this grammar definition that we can use 'd.DATE' when referring to it in the code. This kind of distribution of different grammar rules to separate grammar files is very handy and makes the code much easier to read and maintain.

The line with the # sign is a comment line and it is not used in the grammar.

We have seen the definition of the main verbalization rule in verbalizer.tnm above and here we can see how it is exported from verbalize.grm on the last line. The 'export' statement is an important statement because only definitions exported will be accessible by external fst archives or programs as VoiceData.

> All text files with special symbols used with VoiceData must be saved in UTF-8 encoding without signature! The GrammarEditor saves all text files containing UTF-8 characters automatically in this format!

The Normalization Flow

Because understanding what is happening during the normalization is important, in order to be able to develop better grammars, the normalization process flow will be explained in the next sections.

After the input text is segmented into sentences there are two main steps in the normalization process:

1. **Tokenization/Classification.**
2. **Verbalization.**

Tokenization/Classification

This step will tokenize the sentences into a special sequence of tokens with a specific structure. To see how this works let us take a simple sentence:

"He will give me €1000."

This sentence may be tokenized as follows (will depend on the grammar definition):

```
tokens { name: "he" }
tokens { name: "will" }
tokens { name: "give" }
tokens { name: "me" }
tokens { money { currency: "eur" amount { integer_part: "1000"} } }
tokens { name: "." pause_length: PAUSE_LONG phrase_break: true type: PUNCT }
```

Note several things:

- The words which do not belong to semiotic classes (not defined in the grammars for normalization) are just separated and added as tokens.
- Each semiotic class is tokenized according to the definition in the grammar file. E.g. for the tokenization of the money €1000 above

  ```
  tokens { money { currency: "eur" amount { integer_part: "1000"}}}
  ```

  the tokenizer grammar defines this the following way:

  ```
  q = "\"";
  d =  "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";
  func I[expr] { return "" : expr; }
  func D[expr] { return expr : ""; }
  currencies = StringFile['normdata\en\classify\money.tsv'];
  ```

```
money =     I["money { "]
            I["currency: " q]
            currencies
            I[q]
            I[" amount { "]
            I["integer_part: " q]
            d+
            I[q]
            (D["."]
            I[" fractional_part: " q]
            d{2}
            I[q])?
            I["} }"];
```

The variable ‘money’ here above, is the concatenation of several variables. What we call variable here is actually an FST (finite state transducer); remember that all variables in a grammar define or work with FST’s. Each variable is separated with a space but may also be on a new line. The function `I` is defined to insert the parameter of the function into the target variable (Insert). The function `D` is defined to delete the parameter of the function from the target variable (Delete). These functions may be confusing at first site but we are working with FST’s, e.g. the insert function is just inserting the expr parameter into the target FST. The sequence `("" : expr)` actually means that we rewrite the empty string as the contents of the expr variable in the target FST which actually inserts the expr to the FST. Rewrite is a standard FST operation and used frequently. You could for example do the following `("goez" : “goes”)` which would rewrite “goez” in the FST to be “goes” and thus will autocorrect the accidentally wrongly written “goez” to “goes”.

> Chapter 4 contains an extensive introduction to Finite State Transducers (FST). You may find it useful to first read that chapter if you know nothing about FST’s and how they operate.

The first line ‘ `I["money { "]` ‘ adds the string ‘`money {`‘ as part of the tokenization sequence. The second line ‘ `"currency: " q` ’ adds the string ‘currency:’ and a starting quote (q). The third line ‘currencies’ defines possible strings of currencies. ‘currencies’ is a variable and it is defined in an external text file money.tsv as:

```
$       usd

£       gbp

€       eur

…
```

StringFile['normdata\en\classify\money.tsv'] is a standard grammar function which imports the contents of a text file into a variable.

The variable ‘q’ is for inserting a quote into the token sequence (the quote around e.g. "eur"). The backslash \ is used to escape the quote because it is also the symbol for assigning a string. In case you want to insert a pre-defined symbol (e.g. “, +, *, etc.) as a string into a variable (FST) then you must escape it with a backslash!

At line seven in money ‘d+’ means a digit between 0-9 one or more times (the + sign means the occurrence one or more times). Line ‘ d = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"; ‘ defines variable ‘d’ as one of the quoted digits. This is a union FST operation and ‘d’ accepts any of the arguments.

The rest of the definition is for the fractional part of the money, not needed here in this example but may be interesting to understand.

Line ‘ (D["."] ’ starts with a left brace and what is following is optional thus may not be present in the input token (see note below). For example in €10 there is no fractional part but in €10.50 there is. The function `D` deletes the dot from the input token. This is needed because we verbalize the money as ‘ten euros and fifty cents’ and the period (dot) is not needed in the output string.

> Note that when the brace is closed there is a question mark ‘?’ which signals that this part is optional! The question mark is actually an FST operator and it means that take 0 or 1 time the immediately previous object.

Line ‘ d{2} ‘ accepts two consecutive digits as in the above example 50. Please note that this definition of money will not accept a money statement of €10.5 because 5 is just one digit! This means that €10.5 will not be verbalized but outputted as it is. You would need to extend this grammar to be able to handle €10.5.

The rest of the statements are similar to the above explained ones.

Verbalization

The second step in the normalization process is the verbalization. This step gets its input from the former tokenization step as a sequence of tokens as described above.

We will stay with the same example as above but because the verbalization step only handles semiotic classes (the rest is passes through as tokens) we will only look at what is happening to the money expression (€1000). The other words will just be copied into the output string.

The input to this is the following:

```
money { amount { integer_part: "1000"} currency: "eur" }
```

> Note that the order of tokens is changed; it is now the ‘amount’ first and then the “currency”. It is often the case that we want to verbalize items in a different order than how they appear in the input text. For example €1000 would be verbalized as ‘thousand euros’ rather than ‘euros thousand’ as it is actually in the input. It is not always easy to decide at forehand what the order of words will be in the output and for this reason the normalization engine generates and passes all possible orderings to the verbalizer. It is up to the verbalizer grammar to choose which ordering to use!
>
> In this case both ‘ money { amount { integer_part: "1000"} currency: "eur" } ‘ and ‘money { currency: "eur" amount { integer_part: "1000"} } ’ will be available in the verbalizer.

The verbalizer grammar looks like this

```
import 'normdata\en\verbalize\numbers.grm' as n;

q = "\"";
s = " "*;
d =  "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";
currencies = StringFile['normdata\en\verbalize\money.tsv'];

func I[expr] { return "" : expr; }
func D[expr] { return expr : ""; }

kBytes = Optimize[ "[1]" | "[2]" | "[3]" | … | "[255]"];
sigstar = kBytes*;

ins_maj = CDRewrite[I["_maj"], "", "[EOS]", sigstar];
ins_min = CDRewrite[I["_min"], "", "[EOS]", sigstar];
del_zero = CDRewrite[D["0"], "[BOS]", "", sigstar];

money =
  D["money"]
  D["|integer_part:"]
  n.CARDINAL
  D["|currency:"]
  I[" "]
  (ins_maj @ currencies)
 (I[" and "]
  D["|fractional_part:"]
  (del_zero @ n.CARDINAL)
  D["|currency:"]
  I[" "]
  (ins_min @ currencies))?
  D[s "|"];

singulars = ("cents" : "cent") | ("euros" : "euro");
singularize = CDRewrite[singulars, "[BOS]one " | "and one ", "", sigstar];

export MONEY = Optimize[money @ singularize];
```

In this grammar definition we will learn some new grammar elements and functions.

We have seen before that the import statement imports another grammar file and that the 'as' keyword assign a name to this as a variable. The first line imports the grammar 'numbers.grm' which contains the cardinal (one, two, etc.) and ordinal (first, second, etc.) number names. These names are used to replace the numbers in the input tokens. E.g. 1000 will be replaced by ‘thousand’.

The next several lines are the same as in the tokenizer grammar containing the definitions of quote, any number of spaces, digits, currencies, insert and delete functions. Please note that the “ “* statement means that the FST accepts 0 or any number of spaces. The star * symbol means 0 or any number of (this is an FST operator; see Chapter 4 for more details.). We have seen before that the plus + sign means 1 or any number of elements.

The statement ‘ `sigstar = kBytes*;` ‘ is a frequently used statement and it means that the FST called sigstar will accept 0 or any number of occurrences of any by the kBytes variable defined characters.

The line ‘ `kBytes = Optimize[ "[1]" | "[2]" | "[3]" | … | "[255]"];` ‘ defines all characters in the byte range 1 to 255. Please note that all numbers must be present in the grammar but here the range is shortened (…) in order to not use so much space. Note that byte numbers are defined in between [ ] brackets.

> You may find it interesting to read Chapter 5 Grammar Libraries with information on the byte.grm library.

The ‘Optimize’ grammar function optimizes the final concatenated FST. It is not important to know for the purpose of this book how exactly this happens but an FST contains several internal nodes and arcs (may grow to a big size) and it is often very interesting to call the ‘Optimize’ function on concatenated FST’s in order to reduce the size of the final FST and in order to speed up processing (read more about FST’s in Chapter 4).

The next line ‘ `ins_maj = CDRewrite[I["_maj"], "", "[EOS]", sigstar];` ‘ defines a variable ins_maj which will be used to insert the string “_maj” at the end of strings defined by sigstar (any sequence of bytes, as explained above). The statement [EOS] designates the end of a string (right context in FST terms). [BOS] is the beginning of a string (left context in FST terms). ‘ins_min’ does the same but it inserts “_min”. ‘del_zero’ is very similar but instead of inserting it deletes the character ‘0’ from the beginning of the string where it is applied. Please note that the left [BOS] and right [EOS] context should be used separately in all circumstances.

> It is interesting to note that for example the end of the string sign [EOS] designates one character past the last character in a sentence. The last character is most of the time a period, thus [EOS] means just the location after the dot (which actually does not exist). You can combine the [EOS] sign with other symbols. For example to designate the period you would write ‘`.[EOS]`’. More on this later with the CDRewrite function.

The function CDRewrite will be explained later but here it creates a new variable (FST) by inserting “_maj” at the end of any string built from the by sigstar defined characters.

The next is the money verbalizer definition (will still be extended).

```
money =
  D["money"]
  D["|integer_part:"]
  n.CARDINAL
  D["|currency:"]
```

```
     I[" "]
     (ins_maj @ currencies)
    (I[" and "]
     D["|fractional_part:"]
     (del_zero @ n.CARDINAL)
     D["|currency:"]
     I[" "]
     (ins_min @ currencies))?
     D[s "|"];
```

The first thing to note is that the token sequence is similar to that of the tokenizer grammars' but the format is a bit different. There are no curly braces but pipe | symbols as separators between the fields. This would correspond to the following statement:

```
money|integer_part:1000|currency:eur|fractional_part:50|currency:eur|
```

Most of the lines should be clear to you after reading the former sections, if not then please go back and read it again.

The line ‘ `(ins_maj @ currencies)` ’ will use the above discussed ins_maj function and will insert the string “_maj” at the end of the currency symbol. E.g. it will make “eur_maj”. This is needed for the verbalizer in order to know that this part contains the major part of the money expression which is in our case 1000. The line with ‘ `(ins_min @ currencies)` ‘ is similar but it inserts “_min”, e.g. “eur_min”.

The line ‘ `(del_zero @ n.CARDINAL)` ’ will delete zeros from the beginning of the minor part of the money expression. This is needed in order to be able to verbalize e.g. €1000.05 where we want to output ‘five cents’ instead of ‘zero five cents’.

The next are the following lines:

```
singulars = ("cents" : "cent") | ("euros" : "euro");
singularize = CDRewrite[singulars, "[BOS]one " | "and one ", "", sigstar];

export MONEY = Optimize[money @ singularize];
```

These statements are used to singularize the money expression. This is needed because money.tsv (see the StringFile function above to load this file) contains the plural forms of money:

```
eur_maj  euros
eur_min  cents
```

What we do is just rewriting e.g. ‘euros’ to ‘euro’ in the FST. The ‘ `singularize = CDRewrite[singulars, "[BOS]one " | "and one ", "", sigstar];` ’ function is replacing e.g. “euros” with “euro” if the beginning of the string is “one ” or “and one “. In order to apply this function we must define a new variable ‘MONEY’ as ‘ `MONEY = Optimize[money @ singularize];` ’. The statement ‘ `money @ singularize` ’ applies our singularize function.

The ‘export’ statement makes sure that MONEY is accessible to other parts of the system.

> Important to note that the verbalizer definition for each semiotic class only outputs the lowest level element! In case of the semiotic class ‘money’ the definition contains the following:
>
> ```
> money|integer_part:1000|currency:eur|fractional_part:50|currency:eur|
> ```
>
> The output from the classification/tokenization was:
>
> ```
> money{currency: "usd" amount{ integer_part: "3" fractional_part: "50"}}
> ```
>
> The lowest level elements per section are the currency, integer_part and fractional_part. The element ‘amount’ is therefore removed from the verbalization definition! Read more on this in Chapter 3 Semiotic Classes.

Preserve Order

As we have seen before the token sequence is passed to the verbalizer in all possible order combinations and the verbalizer is responsible to choose which order to ouput. In case if we want to define two or more forms of the same semiotic class then we may need to fix the order in which we want the sequence to be normalized otherwise it would not be sure that we get the output sequence we want.

For example in case of English dates we might have “March 10, 2018” and also “10 March, 2018” and we want to read both in the order as they are tokenized “March tenth twenty eighteen” and “the tenth of March twenty eighteen”. The two forms would be tokenized as follows:

```
date { month: "march" day: "10" year: "2018" }
date { day: "10" month: "march" year: "2018" }
```

In case if we just tokenize and verbalize these like this then the verbalizer would get all combinations of both which would be the same for both:

```
date|month:"march"|day:"10"|year:"2018"|
date|month:"march"|year:"2018"|day:"10"|
date|day:"10"|month:"march"|year:"2018"|
date|day:"10"|year:"2018"|month:"march"|
date|year:"2018"|month:"march"|day:"10"|
date|year:"2018"|day:"10"|month:"march"|
```

and the verbalizer would not be able to distinguish between the two forms of input dates.

The solution to this problem is the “preserve_order” tag which will make sure that the tokenizer fixes the order of tokens and only one order of tokens is passed to the verbalizer. In the above example this can be done as follows:

```
date { month: "march" day: "10" year: "2018" preserve_order: true }
date { day: "10" month: "march" year: "2018" preserve_order: true }
```

and in the verbalizer grammar we must delete some fields introduced by the preserve order tag:

```
# Verbalization for MDY as March 10, 2018
mdy =  D["date"]
  D["|month:"]
  month
  I[" the "]
  D["|day:"]
  day
  I[" "]
  D["|year:"]
  year
  D[preserve_order_specs]?
  D["|"];
```

Where ‘preserve_order_specs’ is defined as follows:

```
# Remove these if they occur
field = (kAlpha | "_")+;
preserve_order = "preserve_order:true";
field_order = "field_order:" field;
preserve_order_specs = (preserve_order | field_order)*;
```

Where the first line defines one or more alphanumeric characters or underscore for the field_order field which is inserted by the tokenizer automatically internally. How kAlpha is defined is not important now (see Chapter 5) but it contains all lower and upper case alphanumeric characters. The statement ‘ `D[preserve_order_specs]?` ’ will remove any occurrences of the ‘preserve_order’ or ‘field_order’ tags.

The statement ‘ `(preserve_order | field_order)*` ’ is any of the ‘preserve_order’ or ‘field_order’ tags zero or more times (the union of the two FST’s).

Thanks to the preserve_order tag we can now do the following in the tokenizer to define both forms of date sequences:

```
# Tokenization for MDY:
mdy = I["date { "]
  I["month: " q]
  month
  D[" "+]
  I[q " day: " q]
  day
  D[","]?
  D[" "+]
  I[q " year: " q]
  year
  I[q]
  I[" preserve_order: true"]
  I[" }"];
# Tokenization for DMY:
dmy = I["date { "]
  I["day: " q]
  day
  D[" "+]
  I[q " month: " q]
  month
  D[","]?
  D[" "+]
  I[q " year: " q]
  year
  I[q]
  I[" preserve_order: true"]
  I[" }"];

export DATE = Optimize[mdy | dmy];
```

In the verbalizer we can then do as follows:

```
# Verbalization for MDY
mdy = D["date"]
  D["|month:"]
  month
  I[" the "]
  D["|day:"]
  day
  I[" "]
  D["|year:"]
  year
  D[field_order_specs]?
  D["|"];
# Verbalization for DMY
dmy = D["date"]
```

```
  I["the "]
  D["|day:"]
  day
  I[" of "]
  D["|month:"]
  month
  D["|year:"]
  I[" "]
  year
  D[field_order_specs]?
  D["|"];
export DATE = Optimize[mdy | dmy];
```

Fail Backoff

For in case the verbalization of a semiotic class fails we can define a backoff (fallback) verbalizer which may for example just copy the sequence of characters to the output but may also do other things as for example replace some of the characters.

The next example will demonstrate some of the possibilities of a backoff verbalizer.

```
import 'normdata\en\verbalize\numbers.grm' as n;

func I[expr] { return "" : expr; }
func D[expr] { return expr : ""; }
q = "\"";
s = " "*;

digit = "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9";
DIGIT = Optimize[digit @ n.CARDINAL];
verbatim_mappings = ("$":"dollar") | ("€":"euro") | ("%":"percent");

MAPPINGS = DIGIT | verbatim_mappings;
ins_space = I[" "];
del_verbatim = D["verbatim:" s q];
del_quote = D[q];

export VERBATIM_MARKUP =
        Optimize[del_verbatim (MAPPINGS ins_space)* del_quote];
```

The first line imports the grammar 'numbers.grm' which contains the cardinal (one, two, etc.) and ordinal (first, second, etc.) number names. These names are used to replace the numbers in the input tokens. So if there are some numbers in the failed verbalization then we can just replace them with their name.

Line two and three are the insert and delete functions we have seen before. Line four and five are the definitions of a quote and 0 or any number of spaces. Note that the star (*) means 0 or any number of the string in front of it.

Line six is the definition of a digit. Line seven (DIGIT) is the mapping between a digit and its name (cardinal). The Optimize function is optimizing the resultant FST.

The line verbatim_mappings is the mapping between some symbols as $, € and % and their names. This is just an example and you could even load these symbols and their names from an external tab delimited text file with the StringFile function as we have done before in case of the money.tsv file in which the different money symbols are mapped to their names.

> **Important**: Text files which contain mappings between strings in two columns must be TAB separated!

The line `MAPPINGS = DIGIT | verbatim_mappings;` combines the two mappings of the numbers and verbatim symbols. The pipe (|) symbol is an union operator (MAPPINGS accepts either of the two argument).

The line with `ins_space` is an example how you can define functions to make your grammar more readable. `ins_space` uses the above defined insert function and inserts a space into the markup. `del_quote` deletes a quote from the markup.

The line with `del_verbatim` defines a function which deletes the string "verbatim:" and any number of spaces and a quote from the markup. This is needed because we get the input containing this string as e.g. "verbatim: %" and obviously we only want to verbalize the %.

The last export line exports the VERBATIM_MARKUP which can be added to the other markups in the verbalizer grammar. We have just seen all of the parts of this function above. One interesting thing to note is the `(MAPPINGS ins_space)*` part which accepts an arbitrary number of (0 or more caused by the * symbol) mapped elements (digits or verbatim elements) followed by a space.

> All text files with special symbols used with VoiceData must be saved in UTF-8 encoding without signature! The GrammarEditor saves all text files containing UTF-8 characters automatically in this format!

Using Weights

Grammars are expressed as regular expressions and context-dependent rewrite rules which are then compiled into weighted finite-state transducers in the OpenFst format. In case you are new to Finite State Transducers then read Chapter 4 with an extended Introduction on Finite State Transducers.

Weights introduce a kind of cost along the arcs of the FST’s. For example if there are two arcs with different weights along a path then the arc with the lower weight (lower cost) will be favored if it is accepted. Weights may be positive and also negative.

The symbol “<n>” (where ‘n’ is a positive or negative number) is used to add a weight to an FST, the weight should be after the FST in question e.g. ‘ `fst1 = "abc" <1>; `’. In case we have a second FST e.g. `fst2 = "acb" <2>;` and if both FST’s are accepting then fst1 will be favored because its weight is smaller than the weight on fst1.

Let us examine a simple example in order to see how to apply weights. The grammar below accepts numbers and can distinguish between an integer number which is a year (verbalized as a year) and a simple number.

```
digit = "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9";
insspace = "" : " ";
zero  = "0" : "zero";
delzero = "0" : "";
units = ("1":"one")|("2":"two")|("3":"three")|("4":"four")|
        ("5":"five")|("6":"six")|("7":"seven")|("8":"eight")|
        ("9":"nine");
teens = ("10":"ten")|("11":"eleven")|("12":"twelve")|("13":"thirteen")|
        ("14":"fourteen")|("15":"fifteen")|("16":"sixteen")|
        ("17":"seventeen")|("18":"eighteen")|("19":"nineteen");
decades = ("2":"twenty")|("3":"thirty")|("4":"forty")|("5":"fifty")|
          ("6":"sixty")|("7":"seventy")|("8":"eighty")|("9":"ninety");

read_digit_by_digit = (units | zero) (insspace (units | zero))*;
two_digit_numbers = teens | decades delzero | decades insspace units;

year = ((("1" | "2") digit) @ two_digit_numbers)
       insspace
       ((digit digit) @ two_digit_numbers
        | ("0" : "oh") insspace (digit @ units));

export NUMBERS = Optimize[(year <-1>) | (read_digit_by_digit <1>)];
```

> You can use the ‘**Test Grammar Rules**’ command in VoiceData to test a grammar like this easily. You can also experiment with different grammars. The test will provide you ten or less accepted output strings so you can see the effect of the weights (costs) you apply. The first line from the ‘Original Text’-box will be used as input!

When you run Test Grammar Rules in VoiceData on the above grammar example then you get the following output:

```
*********************************
* Starting Testing Grammar Rule *
*********************************
Evaluating rule: digit
Evaluating rule: insspace
Evaluating rule: zero
Evaluating rule: delzero
Evaluating rule: units
Evaluating rule: teens
Evaluating rule: decades
Evaluating rule: read_digit_by_digit
Evaluating rule: two_digit_numbers
Evaluating rule: year
Evaluating rule: NUMBERS
Output string: twenty eighteen <cost=-1.000000>
Output string: two zero one eight <cost=1.000000>
```

In case you change the weights to ‘ `(year <1>) | (read_digit_by_digit <-1>)` ‘ then the digit by digit reading will be favored (smaller weight) and you will get the following output:

```
…
Evaluating rule: NUMBERS
Output string: two zero one eight <cost=-1.000000>
Output string: twenty eighteen <cost=1.000000>
```

In this case there are two acceptable output strings and the algorithm is choosing the one which has a lower weight. You can of course have cases with even more accepted output string and thus it is important that you set weights in order to make sure that you get the expected result!

You should understand already most of the expressions in the example grammar but explaining some of them can still be helpful.

```
read_digit_by_digit = (units | zero) (insspace (units | zero))* ;
```

The above line first accepts a digit (think about the FST path along which items are accepted or rejected) between 1 and 9 OR 0 and maps the digit to its name e.g. 1 to one. Then it inserts a space and then maps the following digit in the same way as the first one. Due to the star (*) all following digits (any number of digits are accepted between 0 and ∞) are separated by a space and mapped in the same way to their names.

> Read Chapter 4 Introduction To Finite State Transducers about how these FST operations work and about the operator precedence. Note that we segment the expressions with braces () in order to force a specific operator precedence!

```
two_digit_numbers = teens | decades delzero | decades insspace units;
```

The above line defines all acceptable two digit numbers. It accepts a number between 10 to 19 OR one of the 20, 30, 40, 50, 60, 70, 80, 90. Because the decades variable is defined with only the first digit of the two digit number the second part of the expression '`decades delzero`' deletes the 0 first before mapping it to the name. This also means that this part of the expression will only accept two digit numbers which contain a 0 as a second digit.

The third part of the expression will accept two digit numbers as e.g. 21, 51, 76 and will produce twenty one, fifty one, seventy six. Notice that the insspace variable inserts a space between the first digit and the second one in order to get 'twenty one' instead of 'twentyone' as a mapping for 21.

```
year = ((("1" | "2") digit) @ two_digit_numbers)
       insspace
       ((digit digit) @ two_digit_numbers
       | ("0" : "oh") insspace (digit @ units));
```

The above expression defines a number which is a year. It is a bit more complicated than the former one but the logic is the same. First it accepts a digit 1 or 2 followed by a digit between 0 and 9. This will result in the acceptance of a two digit number which we map to its name with the formerly defined two_digit_numbers function. After this we insert a space because we want to read years as for example 'twenty eighteen' for '2018'. Then we accept any two digits and map the result with the two_digit_numbers function. Instead of the former combination (take attention to the placement of the braces for operator precedence!) we also accept a zero (which we replace with 'oh' and a digit which we map to its name. This last part of the expression will accept a number like 2005 and will rewrite it to 'twenty oh five'.

```
export NUMBERS = Optimize[(year <-1>) | (read_digit_by_digit <1>)];
```

The last line produces the grammar rule which will favor a year instead of a non year number by using weights. The Optimize function will optimize the internal representation of the FST that it will be smaller and faster.

> It is important to note that the expressions in the grammars are separated by spaces and ended by a ‘;’. A longer expression may be on several lines. For example the next statement is valid:
>
> ```
> A = Optimize[
>
>         B
>
>         (D |
>
>         C)];
> ```
>
> which is the same as
>
> ```
>  A = Optimize[B (D | C)];
> ```

Using Self Made Functions

You may define functions in your grammars in order to reduce the code needed and to make your grammars more clear and readable. The following two example functions based on [1] will demonstrate how to define functions.

> More info about ‘byte.grm’ can be found in chapter ‘Grammar Libraries’.

```
import 'byte.grm' as bytelib;

# definitions

digit = bytelib.kDigit;

space = bytelib.kSpace;

sigma = bytelib.kBytes;

slash_b = "[BOS]" | (sigma - (bytelib.kAlnum | "_")) | "[EOS]";

sigma_star = sigma*;

ispace = "" : " ";


func spaceify[A, sigma_star] {

  # Given acceptor "A", generates a rewrite transducer which
  # converts "A " to " A ".

  return CDRewrite["" : " ", A, "", sigma_star] @

         CDRewrite["" : " ", "", A, sigma_star];

}


func tokenize_two[A, B, sigma_star] {
```

```
  # Given acceptors A and B, generates a rewrite transducer which
  # converts "AB" to "A B ".

  return CDRewrite["" : " ", A, B, sigma_star] @

         CDRewrite["" : " ", A " " B, "", sigma_star];

}

# And the next two expressions show how to use the above functions.

# Inserts a space before and after @.

block_rule1 = spaceify["\@", sigma_star];


# Separates 's like strings from the other text by inserting a space
# where needed.

left_contraction = sigma - ("'" | " ");

block_rule2 = tokenize_two[left_contraction,

    ("'S" | "'s" | "'M" | "'m" | "'D" | "'d" | "'") slash_b, sigma_star];
```

*Based on [1]

> In case you want to insert a character which is a predefined symbol like @ then you must escape that character with a backslash \.

For additional functions and explanations please read the chapters ‘Symbols & Functions’ and ‘Grammar Libraries’.

Handling UTF-8 Characters

Depending on the character encoding of your MS Windows version the normalization system may or may not handle special characters correctly. The character encoding difficulty in MS Windows is a long lasting ‘tradition’. For this reason you may define for each language in the VoiceData normalization a symbol table. You will only have to do this if you experience problems with normalizing text with special characters.

Special characters may for example be some accented characters (á, ë, Ā, etc.) but may also be language specific symbols as for example Й, Ж, ڜ or even the °, ©, €, £, etc. symbols.

All of these symbols may be represented in the so called UTF-8 character encoding. We have seen before how to indicate in the grammars that a character or string is in UTF-8 encoding. The problem is that MS Windows uses, depending on the region, special character encoding schemes as for example ‘ISO 8859 – 1 Latin 1; Western European(ISO)’, ‘ANSI Central European; Central European (Windows)’, etc. There are a lot of such encoding schemes. With the symbol table we are able to make sure that all symbols are handled correctly regardless of the active MS Windows character encoding scheme.

The special symbol table must be placed in the language root directory. In case of English this would be the ‘normdata/en’ directory. The symbol table must have the name ‘symbols.tnm’. You may also define a second symbol table with the name ‘symbols_backoff.tnm’ which will be used in case of something goes wrong during the normalization process. Special characters will then be replaced with the names defined in ‘symbols_backoff.tnm’ at the end of the normalization process.

The symbols must be defined in a TAB delimited text file with two columns. The first column must be a unique identifier for the symbol and the second the symbol itself. The text file must be saved in UTF-8 encoding without signature.

> All text files with special symbols used with VoiceData must be saved in UTF-8 encoding without signature! The GrammarEditor saves all text files containing UTF-8 characters automatically in this format!

You can see an example symbol table here under:

```
_x1_	€
_x2_	©
_x3_	§
_x4_	®
_x5_	°
_x6_	±
```

And an example of a symbol back-off table:

```
_x1_	euro
_x2_	copyright
_x3_	paragraph
_x4_	registered trademark
_x5_C	grade celsius
_x5_	grade
_x6_	plus minus
```

The unique identifier may not interfere with the grammar system and therefore it is recommended to use identifiers as shown above in the example. These identifiers start with an underscore and also end with an underscore character. There is also a number added to be able to distinguish several groups of identifiers. You may define for example _x1_, _x2_, etc. as one group of identifiers and a second group with _y1_, _y2_, etc.

Implementing the changes in the grammar files

In case you make use of the above explained symbol table then you must also modify the grammar files as follows. For example if you apply the symbol table to the euro and to the ° symbol then you will have to replace the special identifier in the grammar file e.g. as follows:

```
special = ("_x1_" : "EUR") | ("_x5_" : "°".utf8); # map the unique key
letter_def = kAlpha | special; # this line defines all legal symbols
special_replace = CDRewrite[special, "", "", sigma_star]; # rewrite the FST
```

```
…
```

You will have to make these changes before a string containing the special symbol is used in any way in the grammar. For example if the ° symbol is used in a 'measure' grammar section for °C then you will have to make sure that the grammar can identify the "°C" or "_x5_C" as measure! See the example in the next section for more details on this.

The en_ex example grammar

After you install VoiceData you will notice a subdirectory called "en_ex" under the "normdata" directory (normally located in C:\Program Files\VOICEDATA\normdata\en_ex). This is an example English grammar containing all grammar source codes (grm files). In order to be able to use this grammar in VoiceData you will have to rename the current "en" folder to for example "en_final" and the "en_ex" folder to "en". In this way you will be able to use the example grammar as the English grammar in VoiceData. Make sure that you do not delete the full English grammar but rename it that you can restore it later!

The "en_ex" grammar contains a simplified grammar which can be used for experimentation. It also contains all of the examples explained in this book.

The symbol table is located in: "normdata\en_ex\symbols.tnm", the symbol back-off table is located in: "normdata\en_ex\symbols_backoff.tnm".

There are three grammar definitions (grm files) containing an example of the special UTF-8 character handling explained in this section:

- "normdata\en_ex\classify\word.grm"
- "normdata\en_ex\classify\measure.grm"
- "normdata\en_ex\classify\money.grm"

In wrod.grm you can see the following:

```
…
special = ("_e2_" : "ë".utf8);
word  =  u.I["name:  "  u.q]  ((  (b.kNotSpace  |  special)  <1>)+  |  exceptions)
u.I[u.q];
export WORD = Optimize[word];
```

The line `special = ("_e2_" : "ë".utf8);` replaces the unique key "_e2_" with the corresponding special character "ë".
The next line beginning with "word" adds the special character to the collection of accepted characters.
With this small adjustment the grammar can accept the special character in the normalization.

In measure.grm you can see how to do the same for a special character integrated into a semiotic class.

```
…
measure_and_special = measures|("_x5_":"degree")|("_x5_C":"degree celsius");


measure =
  u.I["measure { "]
  u.I[" decimal { "]
  u.I["integer_part: " q]
  d+
  u.I[q]
 (u.D["."]
  u.I[" fractional_part: " q]
  d+
  u.I[q])?
  u.I["}"]
  u.I[" units: " q]
  u.D[" "*]  # We allow spaces between the number and the measure.
  measure_and_special
  u.I[q]
  u.I["}"];


export MEASURE = Optimize[measure];
```

Notice that the 'measures' variable has been replaced with 'measure_and_special' which contains the replacement of the unique keys from the symbol table with the names of the measurement units. Notice also that a special symbol can be combined with any other character as for example here "_x5_C" is the symbol °C. In the symbol table only "_x5_" is defined which is "°".

> Please note that you may not have to work with a symbol table and may not have to do the above. It all depends on which characters you are using and on your MS Windows codepage. In case you do not want to use a symbol table then make sure that the two files "normdata\en_ex\symbols.tnm" and "normdata\en_ex\symbols_backoff.tnm" are not in the root language folder, in this case the folder "normdata\en_ex". VoiceData will automatically use the symbol tables if they exist in the root language folder! But if they exist then you must also make the adjustments in the grammars as explained above!

Chapter 2 Symbols & Functions

Available Symbols & Functions

Keywords & Symbols*

Symbol	Description
=	Assigns the object on the right side to the identifier on the left side
export	Used to specify that an identifier (FST) will be available from the FST archive after compilation of the grammar. Example: `export fst = "abc";`
()	Used to group an expression, breaking normal operator precedence rules.
;	Each statement should be terminated with this character.
#	Indicates a comment till the end of the line.
.byte	Used to explicitly specify that a string should be parsed byte-by-byte for FST arcs. This is the default option when no parse mode is specified. Examples: • `A = “this is parsed in byte mode.”;` • `B = “this is parsed in byte mode.”.byte;`
.utf8	Used to specify that a string should be parsed using UTF8 characters for FST arcs. Example: `A = “this is parsed in utf-8 mode.”.utf8;`
.mysymtab	Used to specify that a string should be parsed by using a symbol table named mysymtab for FST arcs. Example: `A = “this is parsed in symbol table mode.”.my_symbol_table;`
<n>	Used to attach a weight with value n (positive or negative number) to the FST specified. This should be after the FST in question. Example: `fst = "abc" <1>;`
import	Used to import functions and exported FST’s from another grammar file into the current one.
as	Specifies the aliased name that the imported file uses in the current file. Example: `import 'path/to/utils.grm' as utils;`
func	Used to declare and define a function.
[,,]	Used to specify and separate function arguments.

return	Specifies the object to return from the function to the caller. This keyword is invalid in the main body. Within a function, statements after the return are ignored.
*****	Accepts fst 0 or more times. Example: `result = fst*;`
+	Accepts fst 1 or more times. Example: `result = fst+;`
?	Accepts fst 0 or 1 time. Example: `result = fst?;`
{x,y}	Accepts fst at least x but no more than y times. Example: `result = fst{1,3};`

*This table is based on and mirrors [1] but extended.

> You may better understand these symbols and functions if you read Chapter 4 Introduction To Finite State Transducers.

Standard Library Functions, Operations, and Constants*

The following table contains most of the available functions in grammar files with examples.

Function	Description
fst1 fst2 **Concat[fst1, fst2]**	Concatenates the second fst2 to the first fst1, first it expands the two fst's. The simple form of the function (first line) is just listing of the two fst's with a space between them. Right associative (the first fst is followed by the next one and then follows the next one, etc.). Example: `fst = fst1 fst2;`
ConcatDelayed [fst1, fst2]	Concatenates the second fst2 to the first fst1 but does not expand (delayed) the two fst's. Right associative.
fst1 - fst2 **Difference[fst1, fst2]**	"Takes the difference of two FSTs, accepting only those accepted by the first and not the second. The first argument must be an acceptor; the second argument must be an un-weighted, epsilon-free, deterministic acceptor." [1] The simple form (first line) is just listing of the two fst's with a dash (-) between them. Note: read the chapter about the introduction to FST's in case you need more information about some of the terms here above. Example: `fst = fst1 - fst2;`
fst1 @ fst2 **Compose[fst1, fst2]** **Compose[fst1, fst2, 'left' | 'right' | 'both']**	Applies fst2 to fst1 or with other words it composes fst1 with fst2. The extra argument is for sorting of the input and/or output arguments of the FST's. 'left' will sort fst1's output arcs. 'right' will sort fst2's input arcs. 'both' will do both of the former. The function definition without the sorting option will not perform sorting. The simple form (first line) `fst1 @ fst2` is the same as `Compose[fst1, fst2, 'right'].`

	Example: `fst = fst1 @ fst2;`
fst1 | fst2 **Union[fst1, fst2]**	Accepts either of the two argument FST's, first it expands the two FST's.
UnionDelayed[fst1, fst2]	Accepts either of the two argument FST's, it does not expand the two FST's (delayed). Example: `fst = fst1 | fst2;`
fst1 : fst2 **Rewrite[fst1, fst2]**	Rewrites strings matching the first FST to the second. Example: `fst = fst1 : fst2;`
ArcSort[fst, 'input' | 'output']	"Sorts the arcs from all states of an FST either on the input or output. This function uses delayed FST's." [1] Example: `fst = ArcSort[fst1,'input'];`
Connect[fst]	"Connects a FST, removing unreachable states and paths." [1]

CDRewrite[change, left_context, right_context, sigma_star]	This function defines a context-dependent-rewrite rule.
	Where 'change' is a transducer that specifies a mapping between input and output, 'left_context' and 'right_context' are acceptors (define where the function will be applied), 'sigma_star' defines the alphabet over which the rule is to be applied (should be specified as the transitive closure of all characters that one might expect to see in the input). [1]
	Example 1:
	`fst1=CDRewrite["a":"b","","",sigma_star];`
	This example replaces "a" everywhere with "b" where fst1 is applied (see symbol @ above). The empty string "" for the right and left context mean that 'apply the rule at any place in the string'.
	Example 2:
	`fst1=CDRewrite["":" ","",".[EOS]",sigma_star];`
	This example inserts a space before a period at the end of the string. The symbol "[EOS]" means the end of the string and may only be used in the right context. "[BOS]" mean the beginning of the string and may only be used in the left context!
	Example 3:
	`fst1=CDRewrite["":" ","[BOS]A","",sigma_star];`
	This example inserts a space after a latter 'A' at the beginning of the string. If the '[BOS]' symbol would not be there then after any letter 'A' in the string would be inserted a space.
	You can define any combination of characters for the left and right context.

Determinize[fst]	Makes the FST deterministic. For more info read Chapter 4 with introduction to FST's.
RmEpsilon[fst]	Removes all epsilon (label 0) arcs from the argument FST. For more info read Chapter 4 with introduction to FST's.
Expand[fst]	"Explicitly expands the provided FST to VectorFst (if the FST was already expanded, this just does a reference counted copy and thus is fast)." [1]
Invert[fst]	"Inverts the provided FST. This function uses delayed FSTs." [1]
Minimize[fst]	Minimizes the provided FST.
Optimize[fst]	Optimizes the provided FST. It performs various optimizations on the fst: removes epsilon arcs, sums arc weights, makes deterministic and minimizes. The resulting FST is in general more compact and efficient and faster to compile. Example: `fst = Optimize[fst1 @ fst2];`
Project[fst, 'input' | 'output']	Projects the provided FST onto either the input or the output dimension. Projecting to the output means that the function keeps only the output labels. Projecting to the input means that the function keeps only the input labels. For more info read Chapter 4 with introduction to FST.
Reverse[fst]	Reverses the provided FST.
RmWeight[fst]	Removes weights from an FST:
LoadFst['path/to/fst'] **LoadFstFromFar['path/to/far', 'fst_name']**	Loads an FST either directly from a file or by extracting it from a FAR archive.
StringFile['strings_file']	"Loads a file consisting of a list of strings or pairs of strings and compiles it (in byte mode) to an FST that represents the union of those strings ("string1 | string2 | string3 | ..."), can be significantly more efficient for large lists. If the file contains single strings, one per line, then the resulting FST will be an acceptor. If the file contains pairs of tab-separated strings, then the result will be a transducer. The parse mode of the strings may be specified as arguments to the StringFile function" [1] as follows: `StringFile['strings_file', 'byte', ‘utf8’]` In this example the left string is parsed in byte mode and the right string in utf8 mode.

SymbolTable['path/to/symtab']	Loads the symbol table and returns it as an FST. Example symbol table: car auto bike bicycle produces an FST where "car" maps to "auto" and “bike” maps to “bicycle”.

*This table is based on and mirrors [1] but extended.

> For more info on these functions read Chapter 4 with introduction to FST’s!

> There are still some advanced functions for implementing Pushdown Transducers, Multi-Pushdown Transducers, Features and morphological paradigms and Replace Transducers. In case you are interested in using these features then consult the OpenGrm documentation for further info. You can develop grammars without using these advanced and complex to implement features.

Chapter 3 Semiotic Classes

Available Semiotic Classes

Introduction

In the Grammar Data Files Class Specification Section we have seen how semiotic classes are defined for the internal communication (serialization) between the tokenizer and verbalizer.

Semiotic classes are the backbone of the normalization process because they define the main objects we want to normalize in the text. There are several such classes defined internally in the software and it is of course of importance that you know about them.

The Required column indicates if the field variable (field path) is required or optional. Required fields must exist and thus must also be defined in any serialization specification.

> Please note that there are some terms in this chapter like ‘morphosyntactic features’ (inflection and paradigms, for example in “ child’s book ” the addition of “ ’s ” is a morphosyntactic features) or ‘code-switch’ (indicates the change between languages) of which the explanation is not the aim of this book. If you do not know some of the terms then please consult a specialized linguistics source for more information.

Class Abbreviation

Name	abbreviation		
Purpose	The definition of the shortened form of a word or phrase. Will be expanded using morphosyntactic features.		
Example	Jan., Dr., Mr., km, etc.		
Field Paths	**Path**	**Type**	**Required**
	abbreviation.text	string	Y
	abbreviation.morphosyntactic_features	string	N
	abbreviation.code_switch	string	N

Example Serialization Specification	`class_spec {`
	`  semiotic_class: "abbreviation"`
	`  style_spec {`
	`    record_spec {`
	`      field_path: "abbreviation.text"`
	`    }`
	`    required_fields: "abbreviation.text"`
	`  }`
	`}`		

Class Cardinal

<table>
<tr><th>Name</th><td colspan="3">cardinal</td></tr>
<tr><th>Purpose</th><td colspan="3">The definition of a cardinal number.</td></tr>
<tr><th>Example</th><td colspan="3">1, 250, 1568, etc.</td></tr>
<tr><th rowspan="5">Field Paths</th><th>Path</th><th>Type</th><th>Required</th></tr>
<tr><td>cardinal.integer</td><td>string</td><td>Y</td></tr>
<tr><td>cardinal.morphosyntactic_features</td><td>string</td><td>N</td></tr>
<tr><td>cardinal.code_switch</td><td>string</td><td>N</td></tr>
<tr><td>cardinal.preserve_order</td><td>boolean</td><td>N</td></tr>
<tr><th>Example Serialization Specification</th><td colspan="3"><pre>
class_spec {
  semiotic_class: "cardinal"
  style_spec {
    record_spec {
      field_path: "cardinal.integer"
    }
    required_fields: "cardinal.integer"
  }
}
</pre></td></tr>
</table>

*Preserve order: read the 'The Normalization Flow Preserve Order' section for more info!

Class Date

Name	date		
Purpose	For the definition of a legal date.		
Example	01/12/2018, Jan. 3, 1980, etc.		
Field Paths	**Path**	**Type**	**Required**
	date.weekday	string	N
	date.day	string	N
	date.month	string	N
	date.year	string	N
	date.era	string	N
	date.morphosyntactic_features	string	N
	date.code_switch	string	N
	date.preserve_order	boolean	N

Example Serialization Specification

```
class_spec {
  semiotic_class: "date"
  style_spec {
    record_spec {
      field_path: "date.day"
    }
    record_spec {
      field_path: "date.month"
    }
    record_spec {
      field_path: "date.year"
    }
  }
  style_spec {
    record_spec {
      field_path: "date.month"
    }
    record_spec {
      field_path: "date.day"
    }
    record_spec {
      field_path: "date.year"
      }
    }
  }
```

* Set at least one of month and year.

Class Decimal

Name	decimal		
Purpose	For the definition of the legal format of a decimal number.		
Example	5.1, 12.65, 365.578, -45.2, 2.54E10, etc.		
Field Paths	**Path**	**Type**	**Required**
	decimal.integer_part	string	N
	decimal.fractional_part	string	N
	decimal.quantity	string	N
	decimal.exponent	string	N
	decimal.negative	boolean	N
	decimal.morphosyntactic_features	string	N
	decimal.code_switch	string	N
	decimal.preserve_order	boolean	N
Example Serialization Specification			

```
class_spec {
  semiotic_class: "decimal"
  style_spec {
    record_spec {
      field_path: "decimal.negative"
    }
    record_spec {
      field_path: "decimal.integer_part"
    }
    record_spec {
      field_path: "decimal.fractional_part"
    }
  }
}
```

* NOTE: decimal.quantity is an optional million/billion etc, as an integer.

Class Electronic

<table>
<tr><th>Name</th><td colspan="3">electronic</td></tr>
<tr><th>Purpose</th><td colspan="3">For the definition of a legal electronic items such as URLs, email addresses, etc. “The full schema for URLs, which email addresses can effectively be seen as a subset of, is:
<b>protocol://username:password@domain:port/path?query_string#fragment_id</b>
Hence populating just username and domain will read as an email address.” [1]</td></tr>
<tr><th>Example</th><td colspan="3">https://ai-toolkit.blogspot.com, https://mydomain.com:80, etc.</td></tr>
<tr><th rowspan="12">Field Paths</th><th>Path</th><th>Type</th><th>Required</th></tr>
<tr><td>electronic.protocol</td><td>string</td><td>N</td></tr>
<tr><td>electronic.username</td><td>string</td><td>N</td></tr>
<tr><td>electronic.password</td><td>string</td><td>N</td></tr>
<tr><td>electronic.domain</td><td>string</td><td>N</td></tr>
<tr><td>electronic.path</td><td>string</td><td>N</td></tr>
<tr><td>electronic.query_string</td><td>string</td><td>N</td></tr>
<tr><td>electronic.fragment_id</td><td>string</td><td>N</td></tr>
<tr><td>electronic.port</td><td>integer</td><td>N</td></tr>
<tr><td>electronic.morphosyntactic_features</td><td>string</td><td>N</td></tr>
<tr><td>electronic.code_switch</td><td>string</td><td>N</td></tr>
<tr><td>electronic.preserve_order</td><td>boolean</td><td>N</td></tr>
<tr><th>Example Serialization Specification</th><td colspan="3"><pre>
class_spec {
  semiotic_class: "electronic"
  style_spec {
    record_spec {
      field_path: "electronic.protocol"
    }
    record_spec {
      field_path: "electronic.path"
    }
    record_spec {
      field_path: "electronic.port"
    }
</pre></td></tr>
</table>

<table>
<tr><td></td><td><pre>
    record_spec {
      field_path: "electronic.query_string"
    }
  }
}
</pre></td></tr>
</table>

Class Fraction

<table>
<tr><th>Name</th><td colspan="3">fraction</td></tr>
<tr><th>Purpose</th><td colspan="3">A number that should be read as a fraction, e.g. "three quarters". The input is specified as a separate numerator and denominator.</td></tr>
<tr><th>Example</th><td colspan="3">3/4, .75, 2/4, .50, etc.</td></tr>
<tr><th rowspan="8">Field Paths</th><th>Path</th><th>Type</th><th>Required</th></tr>
<tr><td>fraction.integer_part</td><td>string</td><td>N</td></tr>
<tr><td>fraction.numerator</td><td>string</td><td>Y</td></tr>
<tr><td>fraction.denominator</td><td>string</td><td>Y</td></tr>
<tr><td>fraction.negative</td><td>boolean</td><td>N</td></tr>
<tr><td>fraction.morphosyntactic_features</td><td>string</td><td>N</td></tr>
<tr><td>fraction.code_switch</td><td>string</td><td>N</td></tr>
<tr><td>fraction.preserve_order</td><td>boolean</td><td>N</td></tr>
<tr><th>Example Serialization Specification</th><td colspan="3"><pre>
class_spec {
  semiotic_class: "fraction"
  style_spec {
    record_spec {
      field_path: "fraction.numerator"
    }
    record_spec {
      field_path: "fraction.denominator"
    }
    required_fields: "fraction.numerator"
    required_fields: "fraction.denominator"
  }
}
</pre></td></tr>
</table>

Class Measure

Name	measure		
Purpose	The definition of the legal format of a measure.		
Example	10.1m, 65%, 25.5km, etc.		
Field Paths	**Path**	**Type**	**Required**
	measure.units	string	N
	measure.decimal.integer_part	string	N
	measure.decimal.fractional_part	string	N
	measure.decimal.quantity	string	N
	measure.decimal.exponent	string	N
	measure.decimal.negative	boolean	N
	measure.decimal.morphosyntactic_features	string	N
	measure.decimal.code_switch	string	N
	measure.decimal.preserve_order	boolean	N
	measure.fraction.integer_part	string	N
	measure.fraction.numerator	string	N
	measure.fraction.denominator	string	N
	measure.fraction.negative	boolean	N
	measure.fraction.morphosyntactic_features	string	N
	measure.fraction.code_switch	string	N
	measure.fraction.preserve_order	boolean	N
	measure.cardinal.integer	string	N
	measure.cardinal.morphosyntactic_features	string	N
	measure.cardinal.code_switch	string	N
	measure.cardinal.preserve_order	boolean	N
	measure.morphosyntactic_features	string	N
	measure.code_switch	string	N
	measure.preserve_order	boolean	N
Example			

Serialization Specification	class_spec {
	semiotic_class: "measure"
	style_spec {
	record_spec {
	field_path: "measure.decimal.integer_part"
	}
	record_spec {
	field_path: "measure.decimal.fractional_part"
	}
	record_spec {
	field_path: "measure.units"
	}
	required_fields: "measure.decimal.integer_part"
	}
	}

* NOTE: you may decide in the serialization specification which second level field you make required. Here we require the "measure.decimal.integer_part".

Class Money

Name	money		
Purpose	For the definition of the legal format of a money expression.		
Example	10.10 €, $65, etc.		
Field Paths	**Path**	**Type**	**Required**
	money.currency	string	Y
	money.amount.integer_part	string	Y/N
	money.amount.fractional_part	string	Y/N
	money.amount.quantity	string	N
	money.amount.exponent	string	N
	money.amount.negative	boolean	N
	money.amount.morphosyntactic_features	string	N
	money.amount.code_switch	string	N
	money.amount.preserve_order	boolean	N
	money.quantity	integer	N
	money.morphosyntactic_features	string	N
	money.code_switch	string	N
	money.preserve_order	boolean	N

Example Serialization Specification

```
class_spec {
  semiotic_class: "money"
  style_spec {
    record_spec {
      field_path: "money.amount.integer_part"
      suffix_spec {
        field_path: "money.currency"
      }
    }
    record_spec {
      field_path: "money.amount.fractional_part"
      suffix_spec {
        field_path: "money.currency"
      }
```

```
          }
        }
      }
```

*NOTE: Y/N in the 'Required' column means that one of the fields must be present. Since it can be either the integer_part or the fractional_part we do not require any of them in the serialization specification.

Class Ordinal

Name	ordinal		
Purpose	The definition of an ordinal number e.g. "first".		
Example	1st, 2nd, 57th, etc.		
Field Paths	**Path**	**Type**	**Required**
	ordinal.integer	string	Y
	ordinal.morphosyntactic_features	string	N
	ordinal.code_switch	string	N
	ordinal.preserve_order	boolean	N

Example Serialization Specification

```
class_spec {
  semiotic_class: "ordinal"
  style_spec {
    record_spec {
      field_path: "ordinal.integer"
    }
    required_fields: "ordinal.integer"
  }
}
```

Class Telephone

Name	telephone			
Purpose	For the definition of a legal telephone number.			
Example	+32 222334455, etc.			
Field Paths	**Path**		**Type**	**Required**
	telephone.number_part	This is a repeated field and at least one should be present!	string	N
	telephone.country_code		string	N
	telephone.extension		string	N
	telephone.morphosyntactic_features		string	N
	telephone.code_switch		string	N
	telephone.preserve_order		Boolean	N

Example Serialization Specification

```
class_spec {
  semiotic_class: "telephone"
  style_spec {
    record_spec {
      field_path: "telephone.country_code"
    }
    record_spec {
      field_path: "telephone.number_part"
    }
    record_spec {
      field_path: "telephone.number_part"
    }
    record_spec {
      field_path: "telephone.number_part"
    }
    required_fields: "telephone.number_part"
  }
}
```

*NOTE: Country code (excluding '+'), e.g. 44 for UK.

Class Time

Name	time		
Purpose	The definition of the legal format of the time.		
Example	12:30, 06:25 CET, etc.		
Field Paths	**Path**	**Type**	**Required**
	time.zone	string	N
	time.hours	integer	N
	time.minutes	integer	N
	time.seconds	integer	N
	time.speak_period	boolean	N
	time.morphosyntactic_features	string	N
	time.code_switch	string	N
	time.preserve_order	boolean	N

Example Serialization Specification

```
class_spec {
  semiotic_class: "time"
  style_spec {
    record_spec {
      field_path: "time.hours"
    }
    record_spec {
      field_path: "time.minutes"
    }
  }
}
```

Chapter 4 Introduction to FST

Introduction to Finite State Transducers

Introduction

Modeling natural language features is a very complex subject. The aim of this chapter is to explain in simple lay terms why we need Finite State Transducers (FST's), how they work and how they relate to the text normalization process. For a more in depth description of natural language processing (NLP) and the related subjects of grammatical modeling (orthography and morphophonology), you will need to read several other books (see the reference section for a not exhaustive selection of books).

The so called regular expressions are the backbone of how we model and characterize text sequences or with other words regular languages (more about this a little bit later). In fact if you know regular expressions already then you know a lot about FST's also. But do not worry if you know nothing about it because the chapter will begin with the explanation of regular expressions.

Based on the knowledge of how regular expressions work we will look at the so called Finite State Automaton (or Automata) (FSA) which is just one step away from the Finite State Transducer.
Finite State Automaton is a kind of mathematical device to implement and visualize regular expressions and an important tool in computational linguistics.

As next it will be explained how an FST relates to an FSA and why we need those kinds of FST expressions in our grammar files (see former chapters) and how they relate to regular expressions.

After reading this chapter you will have a deeper understanding of how those FST expressions in the grammar files work. This chapter will however only touch the surface of the possibilities of using FST's in natural language processing. FST's are a very powerful tool and can be used in the grammatical modeling (orthographical and morphophonological modeling) of natural languages. Or better to say, regular languages, because we only model a subset of a natural language and we call this subset regular language (constructed out of a finite alphabet).

Regular Expressions

A text in any language may contain a sequence of letters, numbers, spaces, punctuations, etc. We call one segment of such a text a string. Regular expressions define how we select one or more strings out of a text. We call this selection process 'search'. In all applications of regular expressions we are searching for strings in a text. In order to be able to distinguish regular expressions from other text we enclose them between slashes '/'. The syntax of regular expressions is a very simple kind of textual programming language.

In the following paragraphs several syntactical rules of regular expressions will be explained. The list is by no means an exhaustive list but only an introduction to regular expressions. The information will be just enough for the purpose of this book.

Search for a word syntax

The simplest regular expression is which is searching for a specific word in a text. For example the regular expressions `/fst/` is searching for any string containing the substring ‘fst’. So if we have a sentence ‘fst is very useful’ then the regular expression `/fst/` will match this sentence (string).

Search for a selection of characters or strings

Square braces [ ] are used to specify several characters to match. It is also possible to combine this with a string you are searching for. For example the regular expression `/[fb]ar/` will match a sentence with both words ‘far’ and ‘bar’, thus both of the following sentences will be mathched ‘the bar is open’ and ‘the shop is far’. The expression [fb] thus means that we accept ‘f’ and also ‘b’ as the first letter.

In order to match all upper case letters you can use [A-Z], all single digits is [0-9], etc. Regular expressions are case sensitive.

Repetition and wild card symbols

The ‘*’ star symbol means zero or more occurrences of the immediately previous character or substring. The ‘+’ plus symbol means one or more occurrences in the same way as in case of the ‘*’. The ‘?’ question mark symbol means that the immediately previous character may be missing (0 or 1 occurrence). The ‘.’ period (dot) symbol means the matching of any character.

For example `/ho*/` will match ‘h’, ‘ho’, ‘hoo’, etc. `/ho+/` will match ‘ho’, ‘hoo’, etc. `/ho?/` will match ‘ho’ and ‘h’. `/f.r/` will match ‘far’, ‘fir’, etc.

> If you have read the former chapters then you may see already a similarity with the FST expressions we have used in the grammar definition files. For example `s = “ “*` is an FST defining zero or more number of spaces.

The union of several strings

We can also choose from different strings to match with the pipe ‘|’ symbol. For example `/far|bar/` matches both ‘far’ and ‘bar’.

> This is exactly the same as how we make the union of two FST’s (see former chapters).

Precedence of operators

The repetition and wild card symbols and the pipe symbol are called operators because they make an operation on two or more objects (strings). The highest precedence has the star, plus and the question mark operators then comes the pipe operator. You can also use parenthesis ( ) to indicate your choice of precedence. Just in the same way as with FST’s.

This short introduction to regular expressions is not an exhaustive list of possibilities (there are much more rules which are not mentioned) but it gives you just enough information to understand how the backbone of the natural language processing (string matching e.g. for normalization of text) is working.

Finite State Automata

Finite State Automata (FSA) is a kind of mathematical device to implement and visualize regular expressions and an important tool in computational linguistics.

The FSA and the strings it matches can be illustrated with a simple diagram.

Inspired by the example in [6] we will draw the Finite State Automata for our former example `/hoo+/`. Remember that this regular expression matches the strings ‘hoo’, ‘hooo’, ‘hoooo’, etc. The ‘+’ sign means one or more ‘o’ characters (the character immediately in front of the plus sign).

Each Finite Automata diagram consists of several nodes (one node per state, we will see a bit later what state means) and links (often called arcs) between the nodes ending in arrows to illustrate the direction of the processing.

An FSA recognizes a set of strings (as the regular expression `/hoo+/` also), thus each FSA can be described with a regular expression and vice-versa.

The FSA for this example can be seen hereunder. It has four states denoted with circles from which one is indicated with a double circle, the final state. The final state is an important state in an FSA because in that state the FSA decides whether it accepts the input string or not.

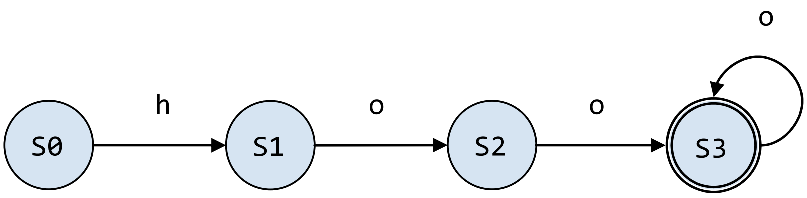


Figure 1 Finite State Automata

The FSA processes the string it receives symbol by symbol and each of these processing steps is ending in a new state. The last or final state is also the accepting state where the FSA decides if it accepts (matches) the incoming string or not.

Each state may also have a transition arc which arrives to itself (self-loop) in order to indicate the acceptance of several symbols (corresponding to the regular expression operator ‘*’ star, zero or more number of symbols).

Let us examine several examples in order to see how our FSA works.

Example 1

Let the input string be 'booo'. The following table indicates what is happening in each state and between them.

State	Decision	Description
S0	h ≠ b	The FSA compares the first symbol of the input 'b' with the first symbol on the arc going out of state S0. Because the symbols do not match the FSA can not proceed to state S1 and fails to recognize, or with other words rejects the input string.

Example 2

Let the input string be 'horse'. The following table indicates what is happening in each state and between them.

State	Decision	Description
S0	h = h	The FSA compares the first symbol of the input 'h' with the first symbol on the arc going out of state S0 'h'. Because the symbols do match the FSA can proceed to state S1 and it also moves to the next input symbol.
S1	o = o	The same as in the former step. The FSA can advance to the next step.
S2	o ≠ r	The third input symbol does not match the letter 'o', which is expected by the FSA, and therefore the machine can not proceed further and rejects the input.

Example 3

Let the input string be 'hoooo'. The following table indicates what is happening in each state and between them.

State	Decision	Description
S0	h = h	OK! Proceed to next state and read next input symbol.
S1	o = o	OK! Proceed to next state and read next input symbol.
S2	o = o	OK! Proceed to next state and read next input symbol.
S3	o = o	OK! Proceed to next state and read next input symbol.
S4	o = o	OK! This is the final input symbol. The FSA accepts the whole input string.

In case of this FSA (`/hoo+/`) example we could say that the different accepted strings describe a regular language containing an infinite number of words as 'hoo', 'hooo', etc. This regular language consists of a finite set of symbols which are 'h' and 'o' (only two). In real situations when we model an existing natural language we have exactly the same logic but of course with much more symbols and rules.

There is one more important property of FSA's which still must be mentioned. This property is the deterministic nature of the FSA. We call an FSA deterministic if it does not need to make decisions about to which state to move on from the current state. Our `/hoo+/` example is a deterministic FSA because it is known in each state without doubt to which state to move on.

If we slightly change our example to `/ho+o/` (notice that the plus sign is moved one symbol to the left) then it becomes a non-deterministic FSA. You can see this easily if you draw the diagram of the Finite State Automata (see image below).

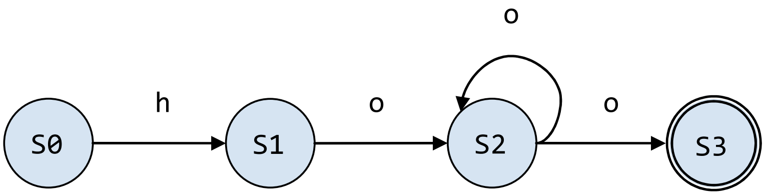


Figure 2 Non-deterministic Finite State Automata with self-loop

In state S2 the machine will have to make a decision, in case it receives a symbol 'o', to stay in state S2 (produces an 'o') or to move on to state S3 which also produces an 'o'. The choice is of course obvious (if there is only one letter 'o' left then it must move to S3 otherwise it must stay in S2) but it has to be made.

The same non-deterministic behavior can be modeled with a so called epsilon (ε) arc instead of a self-loop. The above diagram should then be modified as it is shown on the next image.

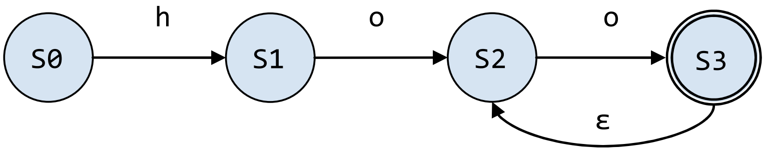


Figure 3 Non-deterministic Finite State Automata with epsilon arc

The epsilon arc means that S3 may not be evaluated and processing can return to S2. This is exactly the same as the FSA on the former image.

Non-deterministic FSA's occur often in natural language processing. According to [6] there are three standard decision making strategies in case of non-deterministic FSA's:

1. **Backup**: "Whenever we come to a choice point, we could put a marker to mark where we were in the input, and what state the automaton was in. Then if it turns out that we took the wrong choice, we could back up and try another path." [6]

2. **Look-ahead**: "We could look ahead in the input to help us decide which path to take." [6]

3. **Parallelism**: "Whenever we come to a choice point, we could look at every alternative path in parallel" [6]

An example for a look-ahead strategy in our grammars (see former chapters) could be for example when we recognize a money expression like 10€. When we process this string then we look-ahead and check if after the number there is a currency sign. If there is then we can process the string as money expression, if not (e.g. 10%) then we do something else (a measure expression). This can be important because in some languages the money expression is spoken (normalized) differently then another type of expressions.

Parallelism could be in a very similar manner for example the checking each of '10€', '10%', '10°C' and then deciding to accept the input or not.

Grammar Modeling (Morphological parsing)

It is interesting and informative to see how to model with FSA's the grammatical problem of finding the plurals of specific English words.

For this we will look at the slightly modified example presented in [6] in which the FSA finds the singular and plural of the words 'cat' and 'goose'. 'Cat' is a regular word and the plural can be formed by just adding an 's' to the end of the word. 'Goose' however is an irregular word and the plural can be formed by changing the so called vowel which then gives 'geese' as the plural form of 'goose'.

The next image shows the simplified FSA diagram of this problem.

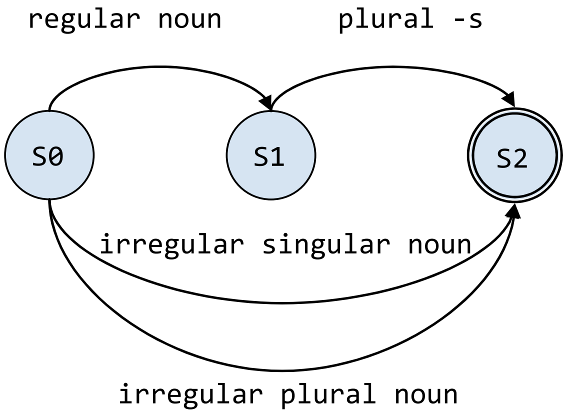


Figure 4 FSA for English nominal inflection. Modified from [6]

If we replace each arc with the sub FSA's containing the symbols (letters) in the studied words 'cat' and 'goose' and also their plural form then we get the composed FSA shown on the following image.

It is easy to see that when the FSA receives the word 'cat' it adds an 's' to the word to create the plural of 'cat'. In case the singular is needed then it passes through the epsilon arc and does not add anything to the string (cat).

In case of the word 'goose' it must decide in state S5 whether the plural or the singular is needed and take the appropriate path.

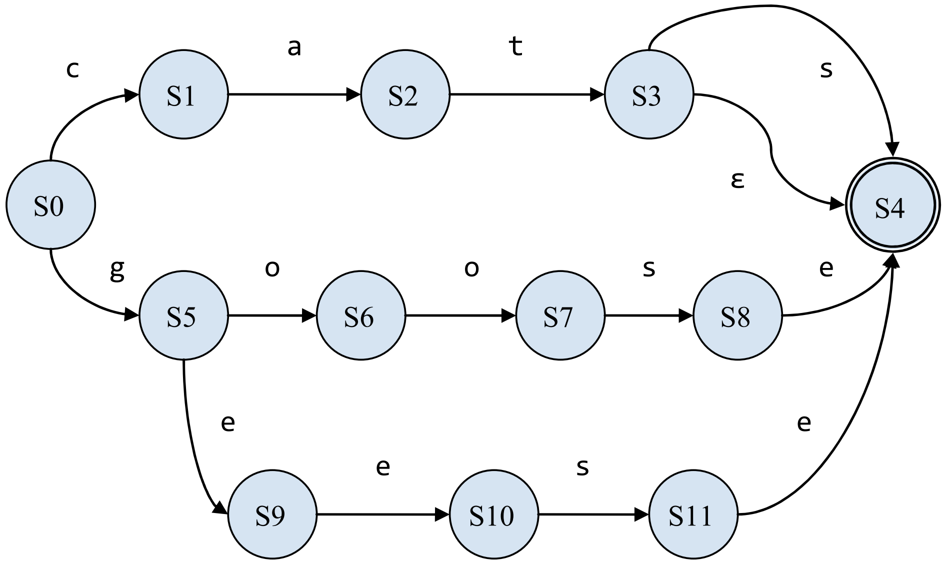


Figure 5 Expanded FSA for English nominal inflection. Modified from [6]

Finite State Transducers (FST's)

The word 'Transducer' indicates that an FST converts symbols from one form to another or with other words it maps between two sets of symbols. An FST is a special form of FSA and therefore they both function very similarly and they can also be visualized similarly.

Mapping between two sets of symbols is indicated by a ':' colon. For example `a:b` means the mapping of 'a' to 'b', or `cycle:bike` maps the word 'cycle' to 'bike'. This is exactly the same notation as we have used in the former chapters to map between two characters or words. You could for example map one word in a specific language to the same word in another language (automatic machine translation).

The visualization of an FST which transforms any number of 'a's to the same number of 'b's can be seen hereunder.

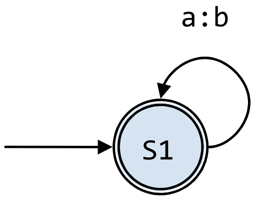


Figure 6 Simple FST for a:b mapping

Everything is the same as with FSA's but now instead of one symbol we have two symbols separated with a colon on each arc. The left side of the colon we call input string and the right side of the colon output string (the input is mapped to the output).

Blackburn & Striegnitz [7] summarize the possible uses of FST’s as follows:

1. **Generation Mode**: The FST in the above example writes a set of ‘a’s as the input string and a set of ‘b’s as the output string. Both set of symbols have the same length.

2. **Recognition or Acceptance Mode**: The FST accepts the input only if the input string has the same number of ‘a’s as the output string has ‘b’s.

3. **Translation Mode (left to right)**: Very intuitively the FST reads ‘a’s from the input and writes the same number of ‘b’s to the output string.

4. **Translation Mode (right to left)**: This is the exact inverse of the former case (and this is also how we call the function which produces the inverse of an FST) and thus it reads ‘b’s from the output and writes ‘a’s to the input. We could also call this reverse translation.

These possible uses indicate already several linguistic fields in which FST’s are used with great success.

We have seen in a former section that in an FSA an epsilon arc can be used to return processing to a former node without evaluating the current state. The same functionality is possible in case of FST’s. Both input and output may be replaced by an epsilon arc as it is shown on the next image.

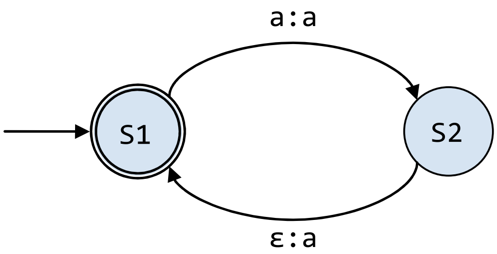
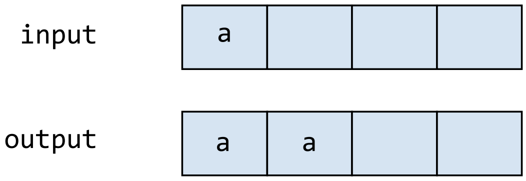


Figure 7 Epsilon arc. Modified from [7]

The above example does the following very intuitively:

1. In Generation Mode it first reads an ‘a’ from the input and writes an ‘a’ also to the output. Then in state S2 it returns to state S1 without doing anything with the input but writing one ‘a’ to the output. S1 is the final state indicated by the double circle. This behavior can also be illustrated with two tapes which symbolize the input and output strings:



2. In Recognition Mode the FST only accepts the input when the output has twice as many ‘a’s than the input.

3. In Translation Mode the FST translates (writes) twice as many ‘a’s to the output than there are ‘a’s in the input.

4. In Reverse Translation Mode (right to left) the FST reads at least two ‘a’s from the output and writes half as many onto the input.

In a very similar manner FST’s can also use string labels containing several symbols.

The following example from [7] illustrates the conversion between English numbers and their names. And this is exactly how the normalization of numbers works in the grammars explained in the former chapters.

The FST for this conversion can be seen hereunder.

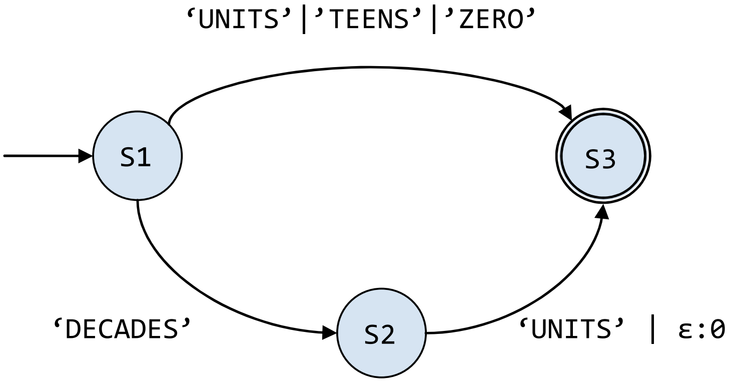


Figure 8 FST for normalizing English numbers. Modified from [7]

The lexicon (collection of words and symbols) for this FST is the following:

UNITS	("1":"one")|("2":"two")|("3":"three")|("4":"four")|("5":"five")|("6":"six")|("7":"seven")|("8":"eight")|("9":"nine")
TEENS	("10":"ten")|("11":"eleven")|("12":"twelve")|("13":"thirteen")|("14":"fourteen")|("15":"fifteen")|("16":"sixteen")|("17":"seventeen")|("18":"eighteen")|("19":"nineteen")
DECADES	("2":"twenty")|("3":"thirty")|("4":"forty")|("5":"fifty")|("6":"sixty")|("7":"seventy")|("8":"eighty")|("9":"ninety")
ZERO	("0" : "zero")

We have seen in Chapter 1 that we form a two digit number (we are making an FST from several FST’s) as follows:

```
TWO_DIGIT_NUMBERS = TEENS | DECADES | DECADES UNITS
```

Please note that for the shake of simplicity the expression here is simplified and not exactly the same as in Chapter 1.

The exact the same behavior or functionality can be seen on the diagram of the FST in the former image. The upper arc accepts numbers between 1-9, 10-19, 0. The lower part of the diagram produces the two digit numbers built from the sub-lexicon DECADES, thus 20, 30, 40, etc. and also 21,22,… 31, … 42, etc. It does this by transitioning from state S2 with an epsilon arc when 20, 30, 40, …etc. and concatenating with the UNITS lexicon when 21, 22, 31, ..etc.

The zero after the epsilon (`ε:0`) means that on the output a zero will be written. This is necessary because the DECADES lexicon contains the decades as 2, 3, 4, etc. thus we make from these numbers 20, 30,…etc. by adding a 0.

Weighted Finite State Transducers (WFST)

The FSA's and FST's can be extended with including weights on the arcs between the nodes. Weights can also be seen as costs on traversing the different paths through an FSA or FST. In case there are several acceptable paths possible (due to the input) through an FST then the path which has the lowest total cost (lowest weight) will be chosen.

In the example below we show the weights next to the arcs between < > symbols. With the same notation as we have used in the former chapters in the grammars for adding weights.

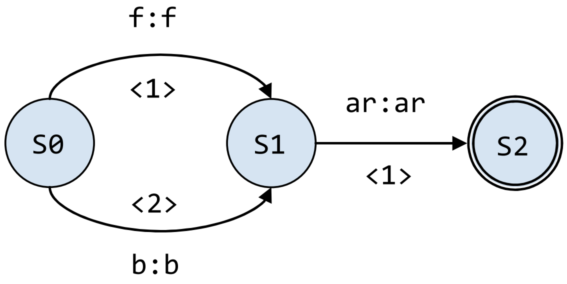


Figure 9 Weighted Finite State Transducer (WFST).

The weights can be any number depending on the software implementation. In our case the weights can even be negative numbers. The aim is to make a weight ordering according to their magnitude.

The FST above will prefer (accept) the word 'far' instead of the word 'bar', whenever both are possible due to the input, because the weight on 'f' is lower than the weight on 'b'.

Operations With FST's

There are several operations possible with (between) FST's as for example inversion (switch the input with the output), composition (connect several FST's in series after each other), projection (extract the input or output depending on the side of the projection), etc.

All of these operations are mentioned already in the former chapters while explaining the composition of the grammar rules for normalization.

In order to make clear what is happening during these operations let us examine an example from [8]. The objective of this example is to derive words by using the prefix 'co' and a simple lexicon containing three English words 'offer', 'design' and 'develop'. The aim is to design an FST which produces the words 'co-offer', 'codesign' and 'codevelop' automatically. The FST should be able to decide when to use a hyphen and when not. This is a typical simplified example of how we use FST's to model grammatical rules (derivational morphology).

We build this FST from a series of sub-FST's which are connected to each other, one after the other. We must do this by writing a series of rules in the form of FST's from the most general rule to the most specific rule [8].

We design this FST with the following three steps:

First Step (FST1): The most general rule

Our first sub-FST (FST1) will contain the most general rule of our case. It transforms an input string beginning with 'co+' followed by any other character to the concatenation of the two as for example in case of 'co+design' it writes 'codesign'. It just removes the '+' sign from the output.

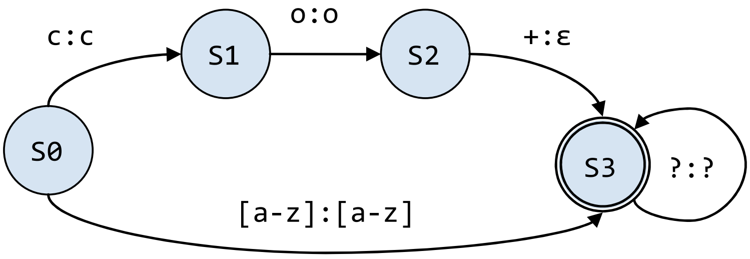


Figure 10 FST1. FST operations example. Modified from [8]

The expression `[a-z]` indicates all letters between 'a' and 'z', in the same way as we have used this in regular expressions before. The expression `+:ε` means that the input accepts '+' and writes nothing (ε) to the output. The expression `?:?` means any character in the input which is written to the output (remember to the regular expression syntax).

The lower arc accepts any input string which does not start with 'co+'. It writes these strings to the output (does nothing with them).

Second Step (FST2): The most specific rule

As next we develop a second sub-FST (FST2) which adds a hyphen when the input word (after 'co+') starts with the letter 'o'. For example in case of an input string 'co+offer' it writes 'co-offer' to the output (it changes the + sign to the - sign). In case the input string does not start with the letter 'o' then the string (e.g. co+design) just passes through the FST (happens nothing to it).

The image below shows the diagram of FST2.

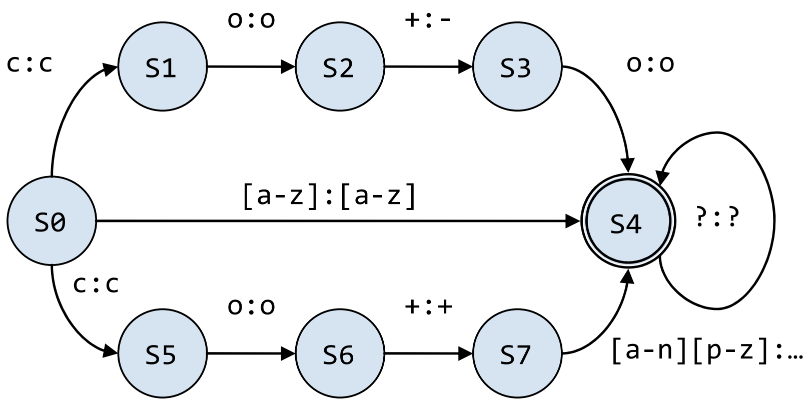


Figure 11 FST2. FST operations example. Modified from [8]

The upper arc path through states S1 to S4 processes 'co+o' and writes 'co-o' to the output. The `?:?` indicates the acceptance of any character after this.

The middle arc accepts any word not starting with 'co+'.

The lower arc path processes 'co+' plus any letter except 'o' ([a-n][p-z]) and then it accepts any letter with `?:?` in state S4, and for example it writes 'co+design' to the output if 'co+design' is the input.

Third Step (FST): The composition of FST1 & FST2

The last step combines the former two FST's into one final FST. We create the combination of the two sub-FST's by the FST operation 'composition' which is also denoted with the following expression:

```
FST = FST2 o FST1
```

Please note that the composition operation applies always first the last FST in a series of FST's to the input.

For example if we have 'co+offer' as input then FST2 produces 'co-offer'. When applying FST1 as next nothing happens since the upper arc in FST1 accepts a string which contains the '+' sign only and through the lower arc just passes everything through without modification.

If the example input would be ‘co+design’ then the output from FST2 would be ‘co+design’ and then FST1 would change this to ‘codesign’ because the upper arc removes the ‘+’ sign on the arc between the states S2 and S3 in FST1.

We could draw the resultant FST as follows (please note that this is already the simplified diagram because the two diagrams are just connected in series, first FST2 and then the output of FST2 is connected to the input of FST1):

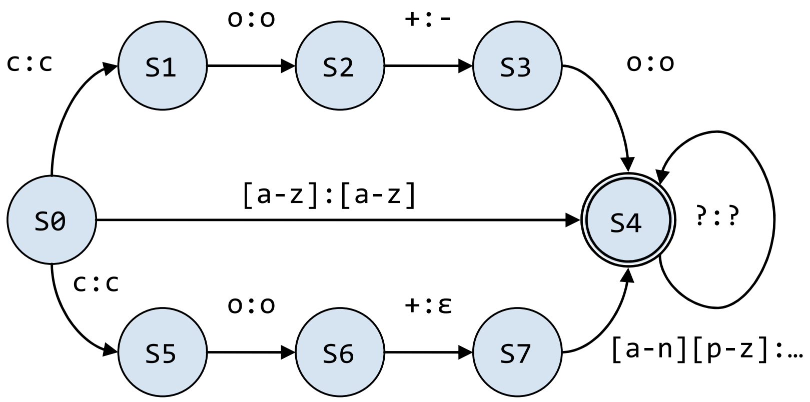


Figure 12 The final FST. FST operations example. Modified from [8]

Conclusion

Modeling natural language features is a very complex subject. The aim of this chapter was to explain in simple lay terms why we need Finite State Transducers (FST’s), how they work and how they relate to the text normalization process. For a more in depth description of natural language processing (NLP) and the related subjects of grammatical modeling (orthography and morphophonology), you will need to read several other books (see the reference section for a not exhaustive selection of books).

After reading this chapter you should have a deeper understanding of how those FST expressions in the grammar files work. We have however just only touched the surface of the possibilities of using FST’s in natural language processing. FST’s are a very powerful tool and can be used in the grammatical modeling (orthographical and morphophonological modeling) of regular languages.

Chapter 5 Grammar Libraries

Available Grammar Libraries

byte.grm

This library from Google is a collection of very useful definitions of ASCII characters.

Contents

```
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
#
# Copyright 2005-2011 Google, Inc.
# Standard constants for ASCII (byte) based strings.  This mirrors the
# functions provided by C/C++'s ctype.h library.
#  Note  that  [0]  is  missing.   Matching  the  string-termination  character
# is kinda weird.

export kBytes = Optimize["[1]" | "[2]" | "[3]" | "[4]" | "[5]" | "[6]" | "[7]"
| "[8]" | "[9]" | "[10]" | "[11]" | "[12]" | "[13]" | "[14]" | "[15]" | "[16]"
|  "[17]"  |  "[18]"  |  "[19]"  |  "[20]"  |  "[21]"  |  "[22]"  |  "[23]"  |  "[24]"
|  "[25]"  |  "[26]"  |  "[27]"  |  "[28]"  |  "[29]"  |  "[30]"  |  "[31]"  |  "[32]"
|  "[33]"  |  "[34]"  |  "[35]"  |  "[36]"  |  "[37]"  |  "[38]"  |  "[39]"  |  "[40]"
|  "[41]"  |  "[42]"  |  "[43]"  |  "[44]"  |  "[45]"  |  "[46]"  |  "[47]"  |  "[48]"
|  "[49]"  |  "[50]"  |  "[51]"  |  "[52]"  |  "[53]"  |  "[54]"  |  "[55]"  |  "[56]"
|  "[57]"  |  "[58]"  |  "[59]"  |  "[60]"  |  "[61]"  |  "[62]"  |  "[63]"  |  "[64]"
|  "[65]"  |  "[66]"  |  "[67]"  |  "[68]"  |  "[69]"  |  "[70]"  |  "[71]"  |  "[72]"
|  "[73]"  |  "[74]"  |  "[75]"  |  "[76]"  |  "[77]"  |  "[78]"  |  "[79]"  |  "[80]"
|  "[81]"  |  "[82]"  |  "[83]"  |  "[84]"  |  "[85]"  |  "[86]"  |  "[87]"  |  "[88]"
|  "[89]"  |  "[90]"  |  "[91]"  |  "[92]"  |  "[93]"  |  "[94]"  |  "[95]"  |  "[96]"
| "[97]"  | "[98]"  | "[99]"  | "[100]"  |"[101]"  | "[102]"  | "[103]"  | "[104]"
|  "[105]"  |  "[106]"  |  "[107]"  |  "[108]"  |  "[109]"  |  "[110]"  |  "[111]"
|  "[112]"  |  "[113]"  |  "[114]"  |  "[115]"  |  "[116]"  |  "[117]"  |  "[118]"
|  "[119]"  |  "[120]"  |  "[121]"  |  "[122]"  |  "[123]"  |  "[124]"  |  "[125]"
|  "[126]"  |  "[127]"  |  "[128]"  |  "[129]"  |  "[130]"  |  "[131]"  |  "[132]"
|  "[133]"  |  "[134]"  |  "[135]"  |  "[136]"  |  "[137]"  |  "[138]"  |  "[139]"
|  "[140]"  |  "[141]"  |  "[142]"  |  "[143]"  |  "[144]"  |  "[145]"  |  "[146]"
|  "[147]"  |  "[148]"  |  "[149]"  |  "[150]"  |  "[151]"  |  "[152]"  |  "[153]"
|  "[154]"  |  "[155]"  |  "[156]"  |  "[157]"  |  "[158]"  |  "[159]"  |  "[160]"
|  "[161]"  |  "[162]"  |  "[163]"  |  "[164]"  |  "[165]"  |  "[166]"  |  "[167]"
|  "[168]"  |  "[169]"  |  "[170]"  |  "[171]"  |  "[172]"  |  "[173]"  |  "[174]"
|  "[175]"  |  "[176]"  |  "[177]"  |  "[178]"  |  "[179]"  |  "[180]"  |  "[181]"
|  "[182]"  |  "[183]"  |  "[184]"  |  "[185]"  |  "[186]"  |  "[187]"  |  "[188]"
|  "[189]"  |  "[190]"  |  "[191]"  |  "[192]"  |  "[193]"  |  "[194]"  |  "[195]"
|  "[196]"  |  "[197]"  |  "[198]"  |  "[199]"  |  "[200]"  |  "[201]"  |  "[202]"
|  "[203]"  |  "[204]"  |  "[205]"  |  "[206]"  |  "[207]"  |  "[208]"  |  "[209]"
|  "[210]"  |  "[211]"  |  "[212]"  |  "[213]"  |  "[214]"  |  "[215]"  |  "[216]"
|  "[217]"  |  "[218]"  |  "[219]"  |  "[220]"  |  "[221]"  |  "[222]"  |  "[223]"
|  "[224]"  |  "[225]"  |  "[226]"  |  "[227]"  |  "[228]"  |  "[229]"  |  "[230]"
|  "[231]"  |  "[232]"  |  "[233]"  |  "[234]"  |  "[235]"  |  "[236]"  |  "[237]"
|  "[238]"  |  "[239]"  |  "[240]"  |  "[241]"  |  "[242]"  |  "[243]"  |  "[244]"
|  "[245]"  |  "[246]"  |  "[247]"  |  "[248]"  |  "[249]"  |  "[250]"  |  "[251]"
| "[252]" | "[253]" | "[254]" | "[255]"];


export kDigit = Optimize["0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" |
"9"];

export kLower = Optimize["a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" |
"j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" |
"w" | "x" | "y" | "z"];

export kUpper = Optimize["A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" |
"J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" |
"W" | "X" | "Y" | "Z"];

export kAlpha = Optimize[kLower | kUpper];

export kAlnum = Optimize[kDigit | kAlpha];

export kSpace = Optimize[" " | "\t" | "\n" | "\r"];

export kNotSpace = Optimize[kBytes - kSpace];

export kPunct = Optimize["!" | "\"" | "#" | "$" | "%" | "&" | "'" | "(" | ")"
| "*" | "+" | "," | "-" | "." | "/" | ":" | ";" | "<" | "=" | ">" | "?" | "@"
| "\[" | "\\" | "\]" | "^" | "_" | "`" | "{" | "|" | "}" | "~"];

export kGraph = Optimize[kAlnum | kPunct];
```

Examples Of Use

Please note that these examples are based on the Google documentation ([1] author: Richard Sproat).

All of the below examples import byte.grm with the following code:

```
import 'byte.grm' as bytelib;
```

```
# Use the transducer defined by kSpace in byte.grm to map a sequence of one or
# more spaces to exactly one space:

reduce_spaces = bytelib.kSpace+ : " ";
```

```
# This deletes a sequence of zero or more spaces:

delspace = bytelib.kSpace* : "";
```

```
# For context-dependent rewrite rules, one must specify the alphabet
# over which the rule is to apply. This should be specified as the transitive
# closure of all characters that one might expect to see in the input. The
# simplest way to do this in general is to allow it to consist of any sequence
# of bytes as below. This is not necessarily the most efficient way to specify
# it. If you know that your input will be much more restricted, then you can
# specify a smaller alphabet, which in turn will yield gains in compilation
# efficiency:

sigma_star = bytelib.kBytes*;
```

```
# This rule illustrates the difference operator. One can specify the
# difference between any pair of regular expressions that specify *acceptors*.
# In this case we specify any punctuation symbol that is not a period:
# NOTE(zso): added the subtraction of the "_" (underscore) character.

punct_not_period = bytelib.kPunct - "." - "_";
```

Using the above example in a context-dependent-rewrite rule:

```
# Here we insert a space, the left context anything at all (hence null) and
# the right context the acceptor specified above as punct_not_period:

separate_punct1 = CDRewrite["" : " ", "", punct_not_period, sigma_star];
```

```
# Here we define a legal word which is 1 or more non-space characters:

word = bytelib.kNotSpace+;


# A sentence consists of a sequence of words interspersed with spaces.
# Initial and final spaces get deleted, and inter-word spaces get reduced
# to exactly one space:

sentence = delspace word (reduce_spaces word)* delspace;


# We can use the above in a final exported FST like this:

export TOKENIZER = Optimize[sentence];
```

Chapter 6 VoiceData UI

VoiceData UI Elements For Normalization

The VoiceData UI is designed to make your work easier and intuitive. There is also a built-in Help in the right sidebar. VoiceData does much more than just normalizing your text. It can read text from images in several languages by using your scanner (you can scan books) or saved images. It can improve your images in several ways which can be very interesting as a first step for text recognition. (It can also do wonders with your family photos.)

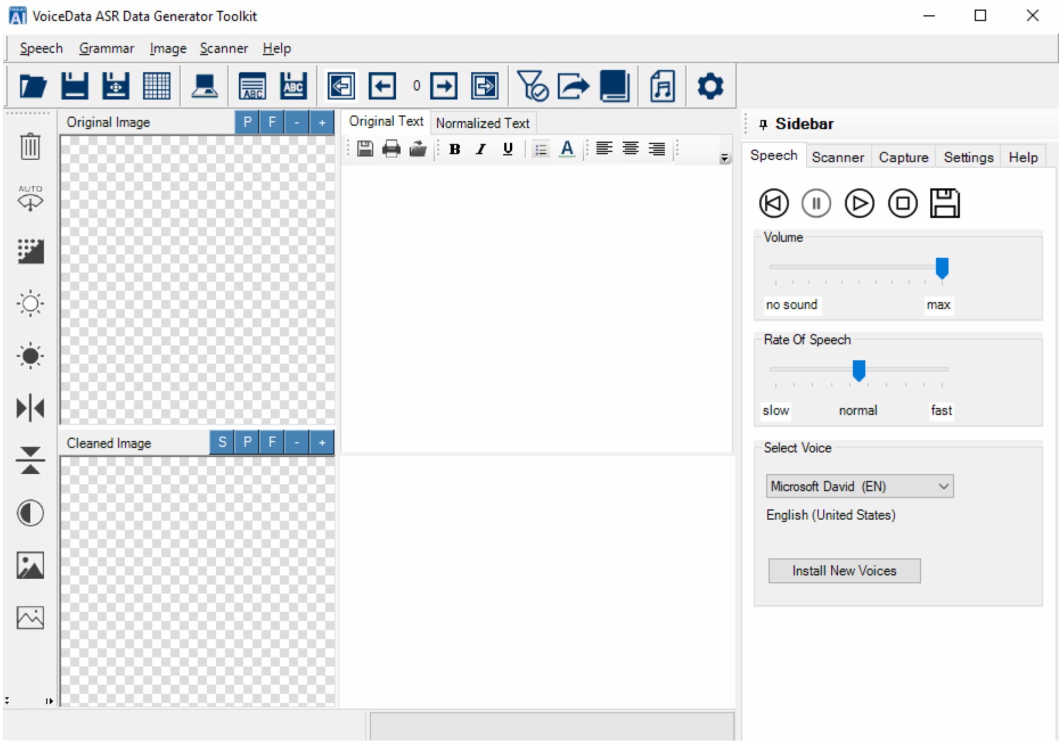
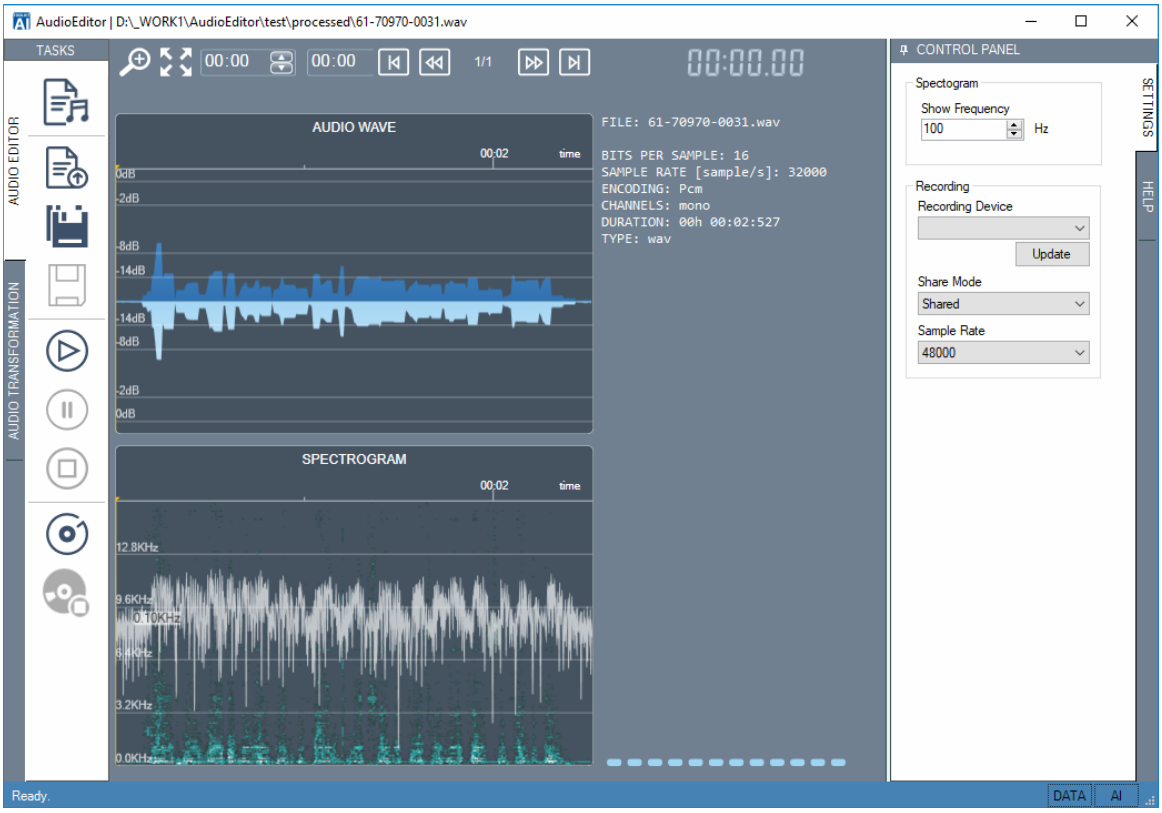


Figure 13 VoiceData UI

VoiceData contains several built-in modules which help you to visualize and edit your audio, export audio recordings together with aligned text transcriptions and even a grammar editor to make your work with grm files easier.

VoiceData is a Microsoft Windows 64bit compatible software package (Windows 7, 8 and 10).

The next two pictures illustrate how the AudioEditor and the GrammarEditor look like.



AudioEditor | D:_WORK1\AudioEditor\test\processed\61-70970-0031.wav

```
FILE: 61-70970-0031.wav

BITS PER SAMPLE: 16
SAMPLE RATE [sample/s]: 32000
ENCODING: Pcm
CHANNELS: mono
DURATION: 00h 00:02:527
TYPE: wav
```

Spectogram

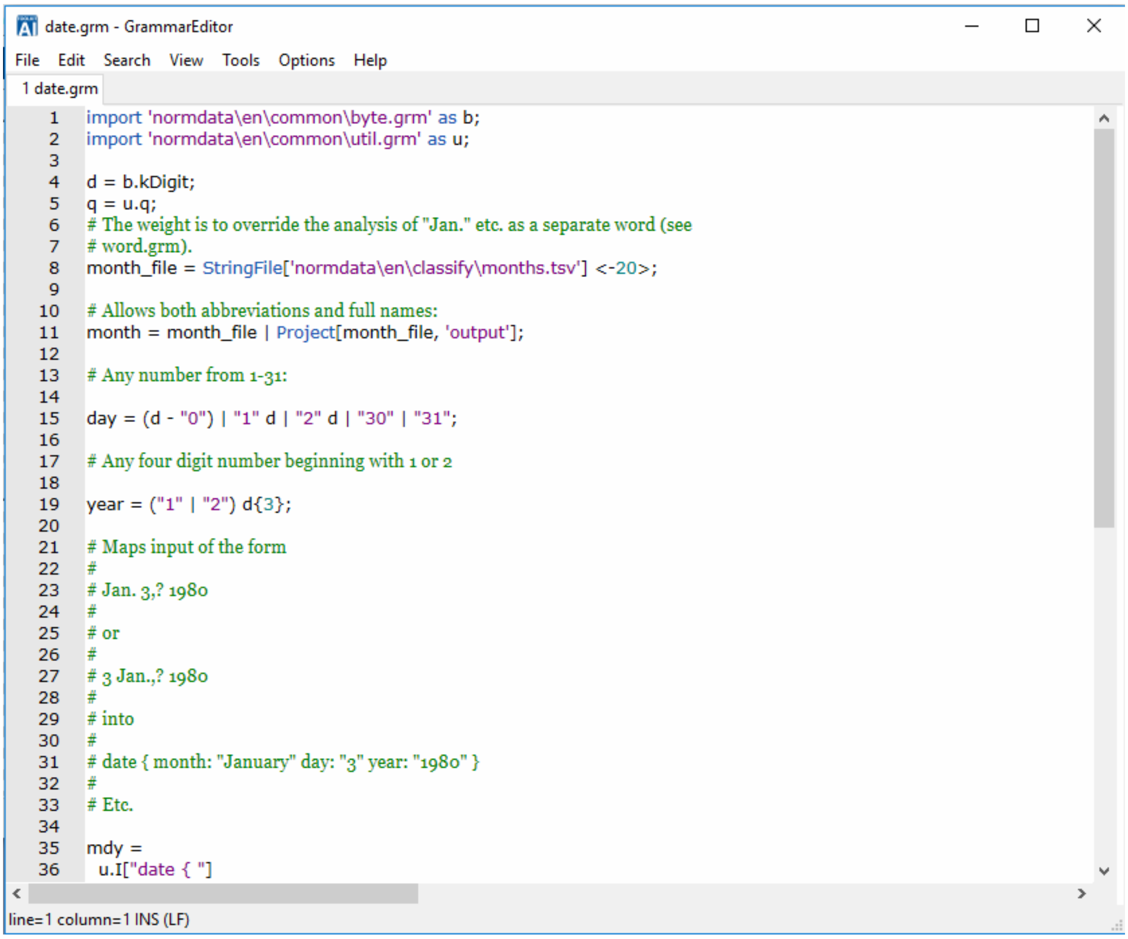
Show Frequency: 100 Hz

Recording

Recording Device:

Share Mode: Shared

Sample Rate: 48000



date.grm - GrammarEditor

```
import 'normdata\en\common\byte.grm' as b;
import 'normdata\en\common\util.grm' as u;

d = b.kDigit;
q = u.q;
# The weight is to override the analysis of "Jan." etc. as a separate word (see
# word.grm).
month_file = StringFile['normdata\en\classify\months.tsv'] <-20>;

# Allows both abbreviations and full names:
month = month_file | Project[month_file, 'output'];

# Any number from 1-31:

day = (d - "0") | "1" d | "2" d | "30" | "31";

# Any four digit number beginning with 1 or 2

year = ("1" | "2") d{3};

# Maps input of the form
#
# Jan. 3,? 1980
#
# or
#
# 3 Jan.,? 1980
#
# into
#
# date { month: "January" day: "3" year: "1980" }
#
# Etc.

mdy =
  u.I["date { "]
```

line=1 column=1 INS (LF)

The most important feature of VoideData is that it can export your text (from several sources and in several languages) as voice recordings per sentence together with the aligned transcription text files which then can be used to train an automatic speech recognition AI model. VoiceData also includes a built-in audio editor which can not only be used to edit, visualize and convert your audio files but also to transform the voice recordings. The audio editor can change the sample rate, the number of channels, can suppress noise, can cancel echo, change the tempo, change the rate, change the pitch frequency and it can also remove audio sections without human voice.

VoiceData has features which makes it possible to completely transfer the audio recording from the base voice to any other types of voice. For example you could change the pitch and the rate of the audio with the built-in audio editor and transfer the adult voice to a child voice. All of these features are working in batch mode and support the use of several processors and processor cores in your computer for very fast processing.

Editing Grammar Files

In the VoiceData Grammar menu with the Edit Grammar command you can start the above shown built-in GrammarEditor which intuitively highlights the grammar format (syntax highlighting) in order to make your work easier. After you are finished with the editing of the file save it and then recompile it together with all linked grammar files in VoiceData.

The GrammarEditor is an advanced text editor and you can search and replace text, change how the text is viewed, visualize line numbers, etc.

Compiling Grammar Files

You have several options for compiling the grammar files into an FST archive which is used during the normalization.

The Recompile All Grammars command will recompile all grammars in the currently selected language indicated by the “grammar_config.tnm” configuration file in the root “normdata” folder of the language (e.g. ‘normdata/en’ for English). On the right sidebar’s Speech tab you can select the desired language in the Select Voice section.

> You can also download new voices and languages. It is however important to note that in case you are downloading new languages you must also develop new grammars for these languages! Please read the first chapter about how the directory structure and configuration files should be setup for a language.

The Recompile All Grammars command will automatically detect any linked/referenced grammar files and will recompile them accordingly.

The Update Compile Grammars command will only recompile grammar files which need recompilation because they or the referenced grammars are changed.

The Compile Grammar command will only compile one grammar file. This will only work if all referenced grammars are already compiled and exist!

Testing Grammar Rules

With the Test Grammar Rules command you can test one or several grammar rules in a grammar file. When you click on this command the program will automatically recompile the necessary grammar files and execute the rule on the first text line in the Original Text box in VoiceData. Make sure that you enter text in the Original Text box first!

> In the Test Rewrite Grammar Rule section on the Setting tab of the right sidebar you must enter one or more grammar rules you want to test! These rules must exist in the grammar file you are testing!

The text log in the middle-bottom section of VoiceData will indicate information about your grammar rules and whether they are tested with success.

Normalizing Text

In order to normalize a text you must first enter the text into the Original Text box or use the text recognition feature on images or with a scanner. You can enter several lines of text.

As a second step select the desired voice/language in the Select Voice section of the Speech tab of the right sidebar. Make sure that a grammar normalization database exists for that language. Please read the first chapter about how the directory structure and configuration files should be setup for a language.

On the right sidebar's Settings tab in the Normalization section check the Show Details option in case you would like to receive details about the normalization steps in the middle-bottom text log. These details can help to solve problems in your grammars.

As a third step make sure that all necessary grammars are compiled and up to date.

As last execute the Normalize Original Text command from the Speech menu. The results will appear in the Normalized Text box. You can further edit the text in the Normalized Text box. At the end (before using the Export feature) you should have only directly readable normalized text in the Normalized Text box.

> **Important**: If the Normalized Text box contains non-normalized text (e.g. 10%) and you export the audio and transcriptions and use them for automatic speech recognition model training then your model will have a lower accuracy and the training of the model may even not work!

In case there is an error during the normalization of a sentence then an "[ERROR NORM]" text will appear in front of the sentence. It is still possible that the normalization was successful but you should check the result. You can always correct any sentence manually!

In case of problems make sure that you switch the "Show Details" option ON mentioned above. The log messages will also contain information about the detected semiotic classes (with all properties) and grammar structure which will give an indication about the location of the error in the grammars.

> As a problem solving step you can use the Test Grammar Rule feature explained in the former section and test specific sections of your grammars.

References

- [0] Ebden and Sproat (2014): Peter Ebden and Richard Sproat. 2014. "The Kestrel TTS Text Normalization System." Journal of Natural Language Engineering.

- [1] OpenGrm Information pages, Copyright © 2008-2018 by the contributing authors. http://www.openfst.org/twiki/bin/view/GRM/ThraxQuickTour, Copyright 2005-2018 Google, Inc., Apache 2.0 license http://www.apache.org/licenses/LICENSE-2.0

- [2] Sparrowhawk documentation, Copyright 2015 and onwards Google, Inc., Apache 2.0, https://github.com/google/sparrowhawk, Copyright 2015 and onwards Google, Inc., Apache 2.0 license http://www.apache.org/licenses/LICENSE-2.0

- [3] Kyle Gorman and Richard Sproat. 2016. "Minimally supervised models for number normalization." Transactions of the Association for Computational Linguistics.

- [4] OpenFst
  www.openfst.org

- [5] VoiceData built-in help. Copyright © 2016 and onwards Zoltan Somogyi (AI-TOOLKIT).

- [6] Speech and Language Processing, An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition, Daniel Jurafsky and James H. Martin, 2008 by Prentice-Hall

- [7] Natural Language Processing Techniques in Prolog, http://cs.union.edu/~striegnk/courses/nlp-with-prolog/html/node13.html, Patrick Blackburn, Kristina Striegnitz, Union College Computer Science department

- [8] Introduction to Finite-State Devices in Natural Language Processing, Emmanuel Roche, Yves Schabes, TR96-13 December 1996. Mitsubishi Electric Research Laboratories.

- [9] Computational Approaches to Morphology and Syntax, Brian Roark and Richard Sproat, Oxford University Press, 2007

- [10] Terry Tai, Wojciech Skut and Richard Sproat. "Thrax: An Open Source Grammar Compiler Built on OpenFst." ASRU 2011, Waikoloa Resort, Hawaii, December.

- [11] Brian Roark, Richard Sproat, Cyril Allauzen, Michael Riley, Jeffrey Sorensen and Terry Tai. 2012. The OpenGrm open-source finite-state grammar software libraries. In Proceedings of the ACL 2012 System Demonstrations, pp. 61-66. http://www.opengrm.org

Automatic Speech Recognition (ASR) Data Generator Toolkit

TOOLKIT AI ARTIFICIAL INTELLIGENCE TOOLKIT

THE APPLICATION OF AI MADE EASY!

VOICEDATA TEXT NORMALIZATION

AUTOMATIC SPEECH RECOGNITION (ASR) DATA GENERATOR TOOLKIT

AUTHOR: ZOLTÁN SOMOGYI

HTTPS://AI-TOOLKIT.BLOGSPOT.COM