Speech Synthesis and Low Resource Languages

Rob Clark
Alexander Gutkin
Martin Jansche
Richard Sproat

Google Research
London & New York

Overview

- Introduction to TTS for low resource languages
- Text Normalization and Linguistic Analysis
- Lexicon and Pronunciation
- Phrasing and Prosody
- Synthesis
- The Festival Speech Synthesis System
- Lab Session

Prerequisites for the Lab Session

Step1. Install Docker:

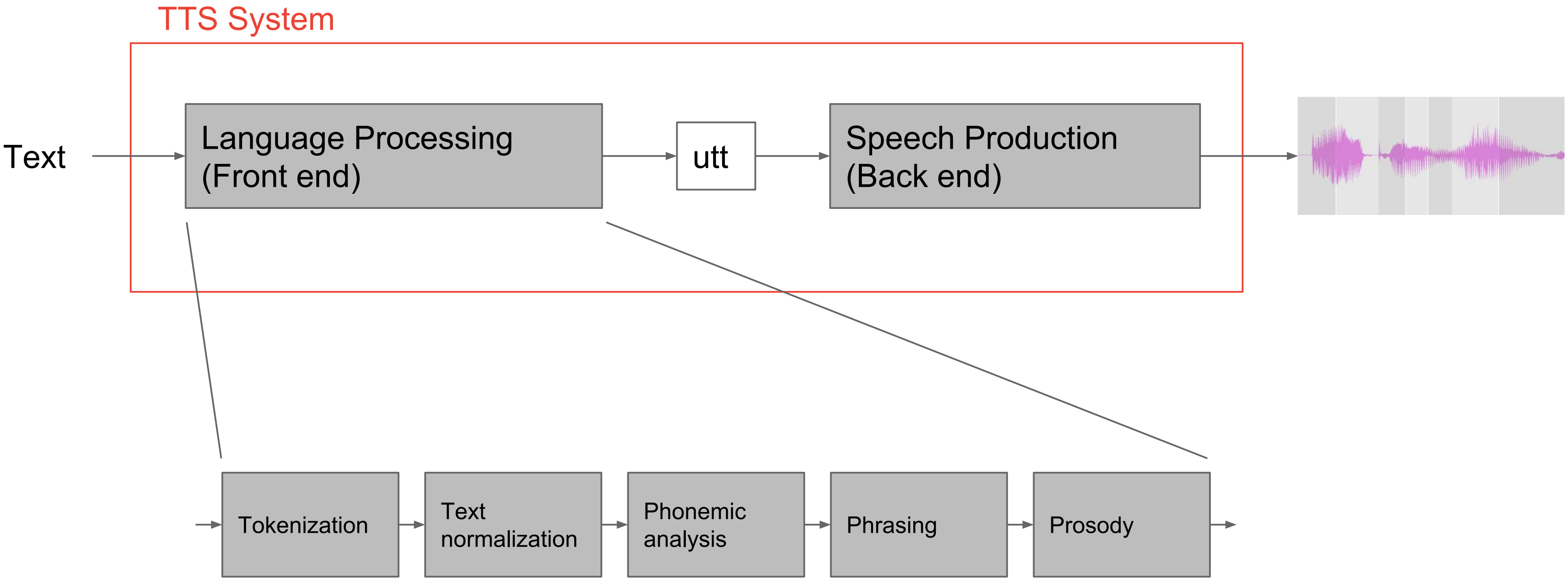
- On GNU/Linux, you minimally need docker-engine.
- On Mac and Windows, you also need docker-machine; best to install the Docker Toolbox.

Step 2. Once you have a working `docker` binary, run this:

```
$ docker pull mjansche/tts-tutorial-sltu2016
```

Introduction

The structure of a TTS system



TTS Frontends

Output:

- Synthesis specification; utterance data structure
- API boundary between analysis frontend and synthesis backend
- Contains all necessary information about the utterance to be synthesized
- Minimally contains a description of the phoneme sequence

Inputs:

- Free text (Text-to-Speech, TTS) (***focus of this tutorial***)
- Structured markup (hybrid)
- Semantic markup (Concept-to-Speech, CTS)

TTS frontends (continued)

Plain text is a wonderful tool for people, but usually a problematic interface choice for software systems.

Think very hard about the systems context in which you want to use TTS. Where does the input text come from? Which aspects of the input text are under your control or influence?

Don’t build a TTS frontend unless you’re absolutely sure that you have to. Typically, a CTS frontend is more robust and simpler to build.

Tokenization / Segmentation

Many Natural Language Processing (NLP) modules operate on word-like units.

Input: Unsegmented sentences.
Task: Find the word-like units (tokens) required by other processing stages.

Challenges:

- Almost no spaces. (Mandarin, Japanese)
- Not enough spaces. (Thai, Lao, Khmer, Burmese)
- Too many spaces. (Vietnamese, Taiwanese POJ)
- Spaces, but not in the right places. (Korean)
- Punctuation. *“He said: ‘Don’t do it.’”*

Text Normalization

Input: Sequence of tokens (the output of tokenization)
Output: Sequence of ordinary words

Example:

- How do you read “123”?
- *It costs $123.*
- *I live at 123 Dr MLK Dr*
- *Lotus 123 was the first “killer app”*

One major focus of this tutorial, more details ahead

Phonemic Analysis

Input: Sequence of ordinary words (the output of text normalization)
Output: Phonemic representation (typically segmental phonemes)

Simplest case: Look up each word in a pronunciation dictionary

Complicating factors:

- What if the word is not in the dictionary?
- Pronunciation of a new word can be guessed from related words, but may differ
- Phonological interactions across word boundaries (external sandhi)

One major focus of this tutorial, more details ahead

Phrasing and Prosody

The macro structure of an utterance, affecting pitch, energy, phonation type, duration, pauses, etc.

Other Text Processing Tasks

- Document layout analysis:
  - Paragraphs
  - Headings
  - Direct quotes
  - Footnotes
- Sentence breaking:
  - “Jill St. John lives on St. Paul St. Paul St. John, her husband, lives on Dr. Martin Luther King Jr. Dr.”
  - “Best. Episode. Ever.”
- Sentence type detection:
  - Yes-No question; alternative question (“tea or coffee?”); interrogative (“wh”) question

    Not always indicated in the orthography (e.g. Japanese)
  - Exclamation (“What was I thinking!”, “To the lighthouse!”)
  - Trailing off, incomplete sentence (“You’re not supposed to… At any rate…”)

Other Text Processing Tasks (continued)

- Breaking of compound words:
  - German: *Rinderkennzeichnungs- und Rindfleischetikettierungsüberwachungsaufgabenübertragungsgesetz* (RkReÜAÜG) (law concerning the delegation of responsibilities for the supervision of the labeling of beef)
  - The compound word is typically not in a dictionary, but all the parts are.
- Diacritic restoration:
  - Diacritics are often hard to type, but text without diacritics is understandable to native readers.
  - E.g. French, Vietnamese, Taiwanese
- Transliteration and non-standard orthography:
  - On-line chat, blog comments may be in Latin script if no other input method is available (e.g. in Arabic, Hindi, etc.)
  - Languages on Twitter (e.g. indigenoustweets.com) may not have standardized writing systems (many regional languages, often with millions of speakers); in fact most languages don't.

TTS Backends

Input: Synthesis specification; utterance datatype (the output of the frontend)
Output: Waveform

Implementations:

- Concatenative synthesis
  - Diphone synthesis
  - Unit-selection synthesis
- Parametric synthesis
  - Hidden Markov models
  - Classification/regression trees
  - Neural network models

One major focus of this tutorial, more details ahead

Low resource languages

“Low resource” is the **default** situation for language data.

Situations where all required data exist are rare and usually artificial.

Can happen:

- In pedagogical settings (**including the lab session of this tutorial**)
- When controlled conditions are required:
  - Competitions
  - Shared tasks in conferences and workshops
  - Benchmarks

Most benchmark datasets (even those for Machine Translation) are not simultaneously large and linguistically diverse.

TTS for low resource languages

Types of data needed:

- Recorded speech (clean, “dry” recordings)
- Unambiguous word-level transcription of recorded prompts
- Pronunciation lexicon
- Phoneme inventory and general linguistic description
- Text normalization examples, test data
- Representative input for evaluation
- Any other data for building frontend analysis modules

Computing resource required are small to modest. Can easily build simple parametric voices on a laptop and deploy them on a phone.

TTS for low resource languages (continued)

Amounts of data needed:

- Depends heavily on implementation and language details and on goals
- 1000 to 3000 recorded sentences for parametric synthesis backend
- Many hours of clean recorded speech for unit selection synthesis backend
- Phonemic transcription of vocabulary; details depend heavily on the language
  10- 000 to 20- 000 words transcribed to start, then estimate learning curve
- Additional data needs depend very heavily on the language:
  - May need tens or hundreds of thousand of segmented sentences to build a segmenter/tokenizer
  - May need thousands of tagged sentences to build a part-of-speech tagger

Where to get data

Assume that you'll have to do your own data acquisition.

- Market for language resources is small and not very efficient.
- Recordings strongly influence the sound of the finished voice.
  - High-quality recordings are often proprietary and not even for sale.
  - Free and open-source recordings are often of decent but not excellent quality.
  - Unit-selection requires single-speaker recordings; can use multi-speaker for parametric backend.
- Availability of pronunciation dictionaries varies considerably:
  - For French, Dutch, and English, large commercial dictionaries are for sale
  - For Icelandic and Bengali, large open-source dictionaries are available
  - For the 11 official languages of South Africa, mid-size open-source dictionaries are available
  - For Javanese, we are not aware of any electronic pronunciation dictionaries
  - Coverage of available dictionaries is typically insufficient

Where to get data (continued)

Recordings for Project Unison:

- Start with ~5000 written sentences to use as prompts for recordings:
  - Review by native speakers to eliminate ungrammatical, offensive, confusing, or otherwise problematic candidate sentences
  - As much as possible, discard or disambiguate ambiguous input
- Record ~10 speakers, 250-350 recordings each
  - Not all sessions will be usable
  - Not all recordings will be usable
  - Built-in safety margin

Still need to bootstrap text normalization and lexicon.

The Festival TTS system

- Open source, started life in 1996 at the University of Edinburgh
- Flexible front end
- Choice of different back ends
  - Original diphone (Unisyn)
  - Limited Domain unit selection (CLUNITS)
  - HTS parametric
  - Clustergen parametric
  - General unit selection (Multisyn)
  - Hybrid DNN/unit selection (unreleased as of May 2016)
- Embedded scheme (lisp) interpreter! 😱

Text Normalization

Text normalization: what is it?

- Conversion of "non-standard" words (NSWs) into ordinary words.
- NSWs, some examples:
    - Numbers written as digits, etc: 324, 5.25E7
    - Dates:
        - 3/24/2009,
        - 2009年3月24日,
        - March 24, 2009
    - Times: 3:27, 3h27, 3時27分
    - Currency amounts: *$1.25, 40万円*
    - Measure phrases: *35kg*
    - Abbreviations: *St., Morningside Hgts*
    - Paul Taylor (2009) refers to many of these as "semiotic classes"
    - *The reading of these is so obvious to competent speakers that they are often surprised when a TTS system gets them wrong*
        - *Robert Walpole, 1st Earl of Orford, KG, KB, PC*

Two components of text normalization

- Given a string of characters in a text, what is the (reasonable) set of possible actual words (or word sequences) that might correspond to it.
- Which of those is right for the particular context?

A simple illustration?

- How do you read “123”?

A simple illustration?

- How do you read “123”?
- *It costs $123.*

A simple illustration?

- How do you read “123”?
- *It costs $123.*
- *I live at 123 Dr MLK Dr*

A simple illustration?

- How do you read “123”?
- *It costs $123.*
- *I live at 123 Dr MLK Dr*
- *Lotus 123 was the first “killer app”*

Two components of text normalization

- A component that gives you the set of possibilities:
  - *123 = one hundred (and) twenty three*
  - *123 = one twenty three*
  - *123 = one two three*
- A component that tells you which of those is right for the particular context.

Example of finite-state methods in text normalization: digit to number name translation

- Factor digit string:
  - *123* → $\mathbf{1 \cdot 10^2 + 2 \cdot 10^1 + 3}$
- Translate factors into number names:
  - $\mathbf{10^2}$ → *hundred*
  - $\mathbf{2 \cdot 10^1}$ → *twenty*
  - $\mathbf{1 \cdot 10^1 + 3}$ → *thirteen*
- Languages vary on how extensive these lexicons are. Some (e.g. Chinese) have very regular (hence very simple) number name systems; others (e.g. Urdu/Hindi) have a large set of number names with a name for almost every number from 1 to 100.
- Each of these steps can be accomplished with finite-state transducers (FSTs)

Hindi/Urdu number names

1	eik	21	ik-kees	41	ikta-lees	61	ik-shat	81	ik-si
2	dau	22	ba-ees	42	baya-lees	62	ba-shat	82	baya-si
3	teen	23	ta-ees	43	tainta-lees	63	tere-shat	83	tera-si
4	chaar	24	chau-bees	44	chawa-lees	64	chaun-shat	84	chaura-si
5	paanch	25	pach-chees	45	painta-lees	65	paen-shat	85	picha-si
6	chay	26	chab-bees	46	chaya-lees	66	sar-shat / chay-aa-shat	86	chaya-si
7	saath	27	satta-ees	47	santa-lees	67	sataath	87	sata-si
8	aath	28	attha-ees	48	arta-lees	68	athath	88	atha-si
9	nau	29	unat-tees	49	un-chas	69	unat-tar	89	
10	dus	30	tees	50	pa-chas	70	sat-tar	90	navay
11	gyaa-raan	31	ikat-tees	51	ika-vun	71	ikat-tar	91	ikan-vay
12	baa-raan	32	bat-tees	52	ba-vun	72	bahat-tar	92	ban-vay
13	te-raan	33	tain-tees	53	tera-pun	73	tehat-tar	93	teran-vay
14	chau-daan	34	chaun-tees	54	chav-van	74	chohat-tar	94	chauran-vay
15	pand-raan	35	pan-tees	55	pach-pan	75	pagat-tar	95	pichan-vay
16	so-laan	36	chat-tees	56	chap-pan	76	chayat-tar	96	chiyan-vay
17	sat-raan	37	san-tees	57	sata-van	77	satat-tar	97	chatan-vay
18	attha-raan	38	ear-tees	58	atha-van	78	athat-tar	98	athan-vay
19	un-nees	39	unta-lees	59	un-shat	79	una-si	99	ninan-vay
20	bees	40	cha-lees	60	shaat	80	assi	100	saw

Kestrel (Sparrowhawk) basics

- Tokenize and classify the tokens
- Store in internal protobuf format
- (Pass tokenized text through morphosyntactic tagger)
- Verbalize the protobufs (possibly with reordering of constituents)

```
I bought 1lb of zebra for £5
```

```
tokens { name: "I" } tokens { name: "bought" } tokens { measure { decimal { integer_part: "1"
} units: "pound" } } tokens { name: "of" } tokens { name: "zebra" } tokens { name: "for" }
tokens { money { currency: "gbp" amount { integer_part: "5" } } }
```

```
money { amount { integer_part: "5" } currency: "gbp" }
```

```
five pounds
```

Peter Ebden and Richard Sproat. 2015 “The Kestrel TTS text normalization system.” *Natural Language Engineering.*

Side note on development tools

- Our hand-built grammars are constructed using a finite-state grammar development environment that we call Thrax (after Dionysius Thrax).
- This has also been open-sourced: see `http://www.openfst.org/twiki/bin/view/GRM/Thrax`
- The Google TTS text normalization is called *Kestrel* (Ebden & Sproat, 2015)
  - There's also an open-source version of that, called *Sparrowhawk,* which will be released integrated into *Festival*

English measures

- Measures:
    - *1 lb → one pound*
    - *5 lb → five pounds* (but cf. *5 lb weight*)
- Stages:
    - *Classify* input as a measure:

      ```
      measure { decimal { integer_part: "1"  } units: "pound" }
      ```

    - *Verbalize* using a grammar of numbers and a grammar of measures. Rules make sure that singular numbers go with singular measures, etc.

Fragment of measure grammar

```
unit_singular = StringFile[
    'speech/patts2/linguistic/kestrel_grammar/verbalize/en/measure_singular.txt'];
unit_plural = StringFile[
    'speech/patts2/linguistic/kestrel_grammar/verbalize/en/measure_plural.txt'];
combined_forms = StringFile[
    'speech/patts2/linguistic/kestrel_grammar/verbalize/en/measure_combined.txt'];
unit_prefixes = StringFile[
    'speech/patts2/linguistic/kestrel_grammar/verbalize/en/measure_prefixes.txt'];

unit_singular_combined = Optimize[
  ((combined_forms @ (unit_prefixes unit_singular))<-10>) |
  ((unit_prefixes util.ins_space)? unit_singular)
];

unit_plural_combined = Optimize[
  ((combined_forms @ (unit_prefixes unit_plural))<-10>) |
  ((unit_prefixes util.ins_space)? unit_plural)
];
```

English currencies

- Measures:
    - *£1 → one pound*
    - *£5 → five pounds* (but cf. *£5 note*)
- Stages:
    - *Classify* input as a measure:

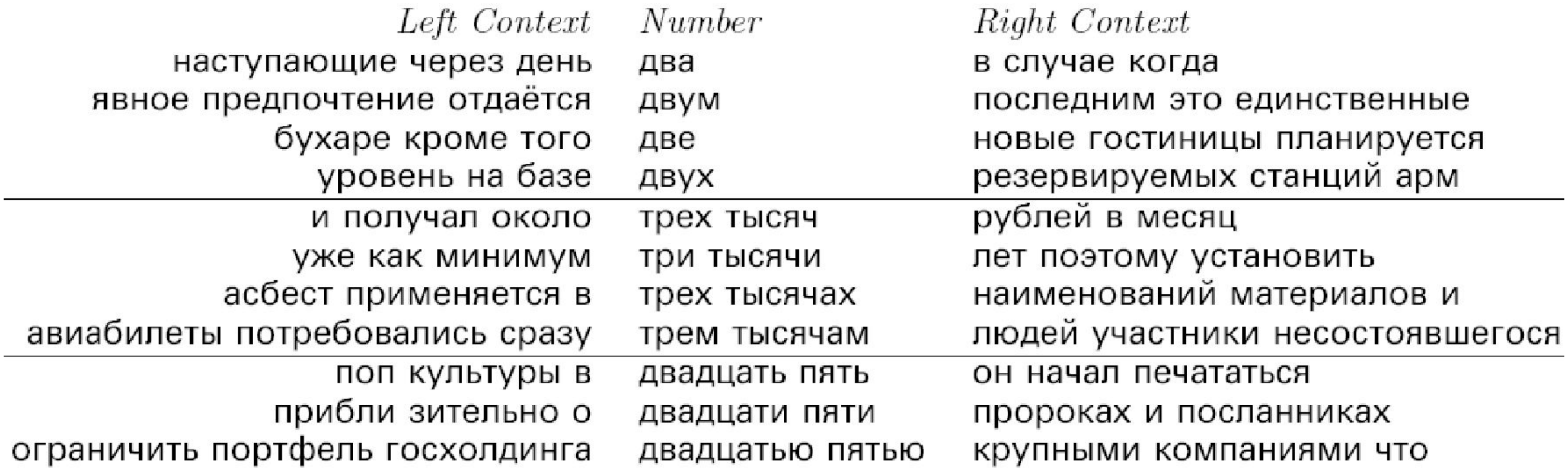
```
money { currency: "gbp" { integer_part: "1"  } }
```

    - *Verbalize* using a grammar of numbers and a grammar of currencies. Rules make sure that singular numbers go with singular currencies, etc.

Russian number names

- Russian distinguishes
  - two numbers (singular, plural),
  - three genders (masculine, feminine, neuter)
  - six cases (nominative, accusative, genitive, dative, prepositional and instrumental).
- Numbers agree in gender with the nouns; noun's case depends on number:
  - Thus **один город** (odin gorod) *one city* has *one* in the masculine nominative/accusative, but
  - **одна собака** (odna sobaka) *one dog*, has *one* in the feminine,
  - **два города** (dva goroda) *two cities*, versus
  - **две собаки** (dve sobaki) *two dogs*.
  - **пять городов** (pjat' gorodov) *five cities*
- In an oblique case, such as the instrumental, the numeral must agree with the noun in case:
  - **в двух шагах** (v dvux shagax) *at two paces*.
- Complex numerals decline in their entirety:
  - **к тремстам тридцати шести часам** (k tremstam tridcati shesti chasam)
    to three hundred and thirty six hours (dative case)
  - **с пятью тысячами пятьюстами семьюдесятью четырьмя рублями**
    (s pjatju tysjachami pjatjustami semjudesjatju chetyr'mja rubljami)
    with five thousand five hundred and seventy four rubles(instrumental case).

Some examples drawn from web sources



Left Context	*Number*	*Right Context*
наступающие через день	два	в случае когда
явное предпочтение отдаётся	двум	последним это единственные
бухаре кроме того	две	новые гостиницы планируется
уровень на базе	двух	резервируемых станций арм
и получал около	трех тысяч	рублей в месяц
уже как минимум	три тысячи	лет поэтому установить
асбест применяется в	трех тысячах	наименований материалов и
авиабилеты потребовались сразу	трем тысячам	людей участники несостоявшегося
поп культуры в	двадцать пять	он начал печататься
прибли зительно о	двадцати пяти	пророках и посланниках
ограничить портфель госхолдинга	двадцатью пятью	крупными компаниями что

Russian measures

- 1кг -> *один килограмм* (odin kilogramm -- nom. sg)
- 3кг -> *три килограмма* (tri kilogramma - gen. sg)
- 5кг -> *пять килограммов* (pjat' kilogrammov - gen. pl)
- 3m^2 -> *три квадратных метра* (tri kvadratnyx metra -- gen. sg, but adjective in gen plur)
- But if the whole phrase needs to be in oblique case ...
  - "from 1 to 3 kilograms":

    от одного килограмма до трех килограммов

    ot odnogo kilogramma do trex kilogrammov

  - "with 5 kilograms"

    с пяти килограммами

    s pjati kilogrammami

Russian measures

- Stages:
  - Classify sequence as a measure as in English
  - Verbalization must take several factors into account
- measure {cardinal {integer: "5"} units: "meter" morphosyntactic_features: "__GEN"}
- Issues:
  - We assume a "morphosyntactic features" slot that gets assigned via a part-of-speech tagger.
  - We must also assume a carefully coordinated set of rules to make sure that the right number form goes with the right measure word form, and vice versa

Russian currencies

- $1 -> *один доллар* (odin dollar -- nom. sg)
- $3 -> *три доллара* (tri dollara -- gen. sg)
- $5 -> *пять долларов* (pjat' dollarov -- gen. pl)
- 3AUD -> *три австралийских доллара* (tri avstralijskix dollara -- gen. sg, but adjective in gen plur)
- But if the whole phrase needs to be in oblique case …
    - "from 1 to 3 dollars":

      от одного доллара до трех долларов

      ot odnogo dollara do trex dollarov
    - “with 5 dollars"

      с пяти долларами

      s pjati dollarami

Russian measures and currencies

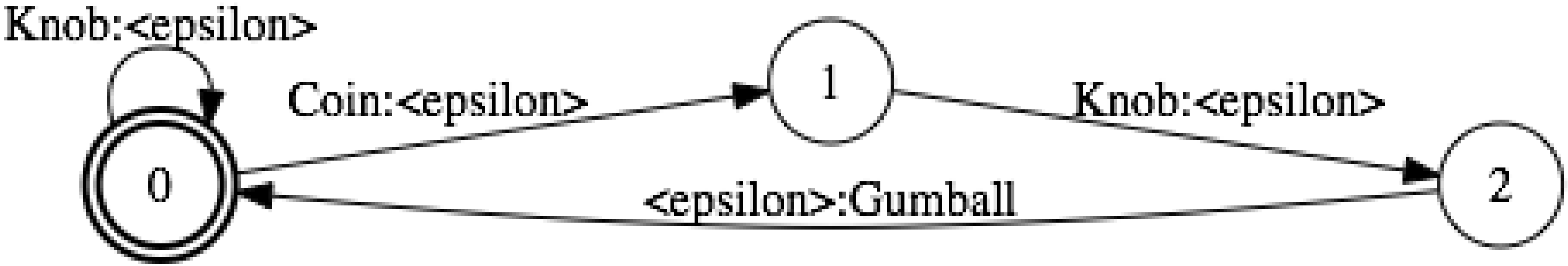
- In English one could write grammars for measures and currencies more or less separately
- In Russian doing so would duplicate a lot of effort, and miss the point that the phenomena are a feature of the language as a whole
- *Good linguistics is good engineering*

Thrax Grammars

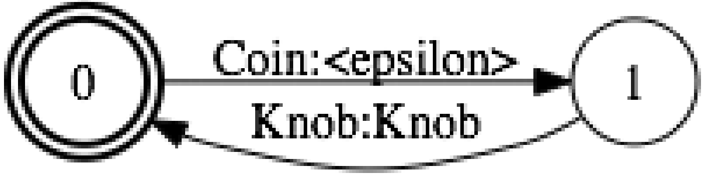
A simple state machine

Credit: etsy.com

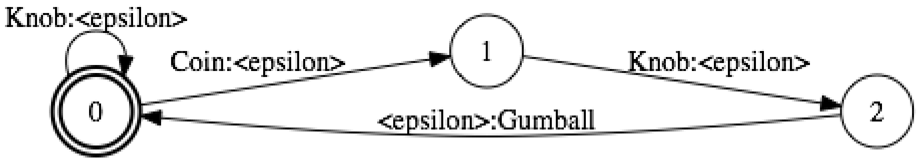
Short digression on weighted finite-state transducers: A simple state machine. Gumball machine *G*



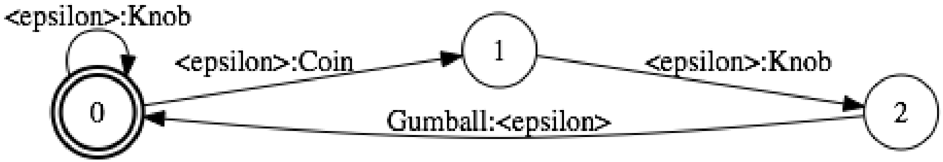
Short digression on weighted finite-state transducers: Making a free version: Modification machine *M*



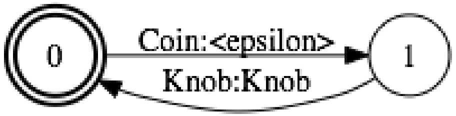
Short digression on weighted finite-state transducers: Making a free version



Inversion



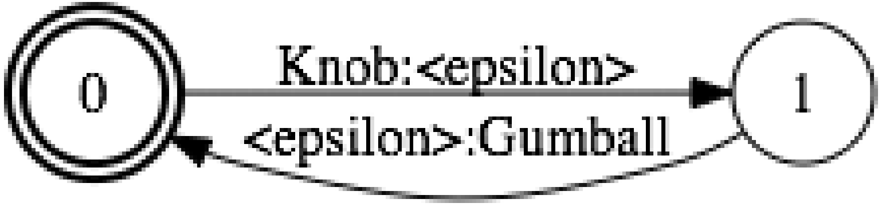
Composition



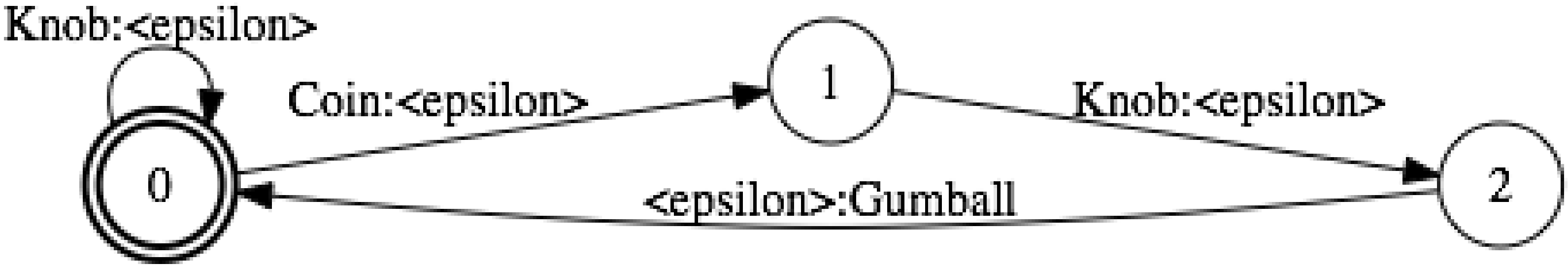
$(G^{-1} \circ M)^{-1}$

Short digression on weighted finite-state transducers: Free version

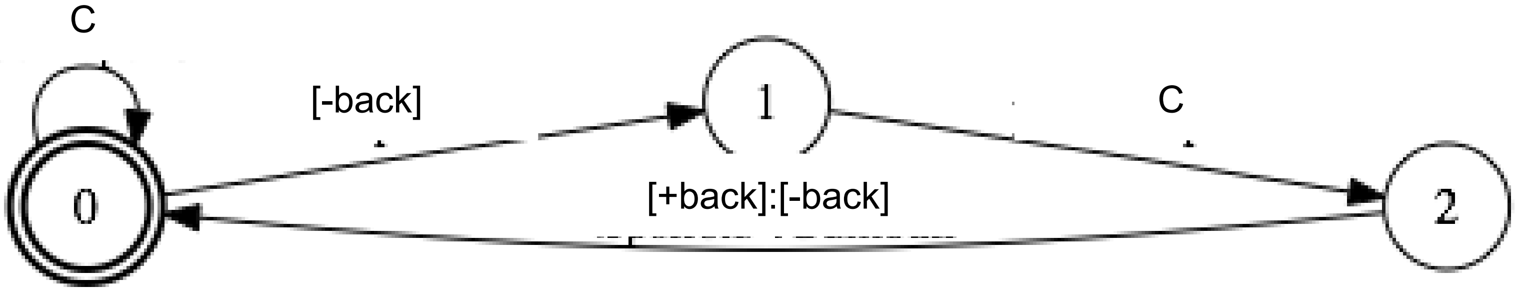
$(G^{-1} \circ M)^{-1} =$



Short digression on weighted finite-state transducers:



Short digression on weighted finite-state transducers:



Short digression on weighted finite-state transducers (WFSTs)

- FSTs can encode *cascades* of rules of the form:
  - φ→ψ / λ __ ρ
- Weights (costs) can be used on arcs to model probabilities, preferences, etc.

Synopsis

- Tools for constructing grammars that can be compiled down into WFSTs.
- Applications include:
  - Text normalization
  - Letter-to-sound rules
  - Transliteration
  - Morphological analysis
  - ...
- Open source version (Thrax) available at http://opengrm.org, with documentation at http://openfst.cs.nyu.edu/twiki/bin/view/GRM/ThraxQuickTour

Overview

- Weighted regular expressions compiled using Thompson construction
- Weighted context-dependent rewrite rules (more on this later)
- Various kinds of optimizations available
- Efficient prefix tree implementation (`StringFile`) for large unions of strings
- Supports symbol tables (byte, utf8, user-provided)
- Supports various semirings (default: tropical, aka standard)

Interlude with Thrax documentation

http://openfst.cs.nyu.edu/twiki/bin/view/GRM/ThraxQuickTour

Digression on context dependent rule compilation

Given a rule of the form:

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

Where φ, ψ, λ and ρ are regular expressions, how can you compile this rule into a transducer?

Example

u →i / i C* ___

u →i / Σ* i C* ___ Σ*

Input: kikukuku

Example

u →i / i C* ___

kik$\boxed{u}$kuku

kikik$\boxed{u}$ku

kikikik$\boxed{u}$

kikikiki

Example

- The output of one application *feeds* the next application
- But we took it for granted that the rule applies left-to-right

Right-to-left application

u →i / i C* ___

kikukuk$\boxed{\text{u}}$

kikuk$\boxed{\text{u}}$ku

kik$\boxed{\text{u}}$kuku

kikikuku

Simultaneous application

u →i / i C* ___

kikukuku

kikikuku

Bookkeeping

- Let’s say we are being extra careful and we want to make sure that we’re applying the rule correctly.
- To help us, we might use some auxiliary annotations, such as angle brackets, to mark contexts, as well as the actual string to be changed.

Bookkeeping: First attempt

u →i / Σ* i C* ___ Σ*

Mark possible right contexts >

k i k i k u p a p u

> > > > > > > > > >

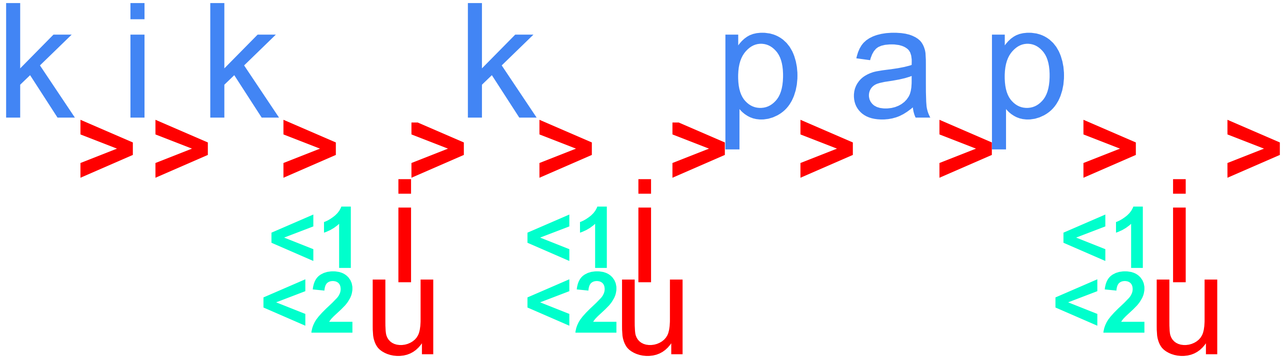
< <

Oops...

This is wrong: we are not allowing for the left-to-right feeding

- Mark the beginnings of the right context (ρ) — that much was correct
- Mark the left edge of all φ that have > on the right with two markers $<_1$ and $<_2$
  - $<_1$ triggers application of change φ to ψ
  - $<_2$ marks *non*-application of the change
- Change φ to ψ between $<_1$ and >
  - Delete >: we don't need it anymore
- We now have a set of possible strings, some with the change, some without
- We now check λ:
  - Allow only strings where $<_1$ is preceded by λ. Then delete $<_1$.
  - Allow only strings where $<_2$ is *not* preceded by λ. Then delete $<_2$.

Bookkeeping: Second attempt



k i k k p a p

> > > > > > > > > >

<1 i <1 i <1 i

<2 u <2 u <2 u

Now in fact...

Each of these can be represented as a transducer:

- A transducer r that inserts > before every ρ
- A transducer f that inserts $<_1$ and $<_2$ before every φ followed by >
- A transducer $replace$ that replaces φ with ψ between $<_1$ and > and also deletes >
- A transducer l_1 that filters out all $<_1$ not preceded by λ and deletes $<_1$
- A transducer l_2 that filters out all $<_2$ preceded by λ and deletes $<_2$

The final rule transducer

A series of cascaded transducers can be composed together to yield a single transducer that computes the same relation as the cascade

$$rule = r \circ f \circ replace \circ l_1 \circ l_2$$

Walk through of the Thrax lab exercise:

- Complete a grammar for Afrikaans numbers
    - But I don’t speak Afrikaans!!!
    - Right, neither do I: But often you have to develop resources for languages you don’t know.
- We provide the factorization piece:
    - *123* → $\mathbf{1 \cdot 10^2 + 2 \cdot 10^1 + 3}$
- You will need to complete the verbalizer portion that maps factored strings to their words,
    - $\mathbf{10^2}$ → *honderd*
    - $\mathbf{2 \cdot 10^1}$ → *twintig*
    - $\mathbf{1 \cdot 10^1 + 3}$ → *dertien*

Walk through of the Thrax lab exercise:

- This includes proper placement of *en* (‘and’) and deletion of *een* (‘one’) before *honderd* (‘hundred’):
  - 130 → *honderd* ***en*** *dertig*
  - 100 → *honderd* (not *een honderd*)
- Included in the data are test examples: you should verify that you get the same answer as what’s in the tests.

Short list of references on text normalization

1. Sproat, R.; Black, A.; Chen, S.; Kumar, S.; Ostendorf, M.; Richards, C. 2001. "Normalization of non-standard words." *Computer Speech and Language* **15**; 287–333.
2. Richard Sproat, 2010. "Lightly Supervised Learning of Text Normalization: Russian Number Names," *IEEE Workshop on Spoken Language Technology*, Berkeley, CA.
3. Deana L. Pennell and Yang Liu. 2011. "A Character-Level Machine Translation Approach for Normalization of SMS Abbreviations." *IJCNLP.*
4. Brian Roark, Michael Riley, Cyril Allauzen, Terry Tai and Richard Sproat. 2012. "The OpenGrm open-source finite-state grammar software libraries". *ACL 2012,* Jeju Island, Korea.
5. Yi Yang and Jacob Eisenstein. 2013. “A Log-Linear Model for Unsupervised Text Normalization.” EMNLP 2013
6. Peter Ebden and Richard Sproat. 2015. "The Kestrel TTS Text Normalization System." *Journal of Natural Language Engineering*.

See also a course I co-taught a few years ago.

Phonemic Analysis: Lexicons and Pronunciation

Phonemic Analysis

Input: Sequence of ordinary words (the output of text normalization)
Output: Phonemic representation (typically segmental phonemes)

Simplest case: Look up each word in a pronunciation dictionary

Complicating factors:

- Human factors
- What if the word is not in the dictionary?
- Pronunciation of a new word can be guessed from related words, but may differ
- Phonological interactions across word boundaries

An English example

Input: it costs one hundred and twenty three dollars

it	ih1 t
costs	k aa1 s t s
one	w ah1 n
hundred	hh ah1 n d r ax d
and	ae1 n d
twenty	t w eh1 n t iy0
three	th r iy1
dollars	d aa1 l er0 z

Source: CMUdict 0.4

An English example

Input: it costs one hundred and twenty three dollars

Output:
```
ih1 t # k aa1 s t s # w ah1 n # hh ah1 n d r ax d # ae1 n d
# t w eh1 n t iy0 # th r iy1 # d aa1 l er0 z
```

Why and when does this work?

The output of text normalization consists exclusively of ordinary words.

We have not fully defined what we mean by “ordinary words”.

Working definition: Ordinary words exhibit a regular, common correspondence between their spelling and their pronunciation.

The correspondence need not be simple. But it needs to give sufficient clues about the intended pronunciation to allow literate native speakers to figure out what sound sequence was intended.

Spelling / Sound Correspondence

a	n	d
ae1	n	d

t	w	e	n	t	y
t	w	eh1	n	t	iy0

Spelling / Sound Correspondence

a	n	d
ae1	n	d

t	w	e	n	t	y
t	w	eh1	n	t	iy0

t h	r	e e
th	r	iy1

o	n	e
w ah1	n	

Spelling / Sound Correspondence

a	n	d
ae1	n	d

c	o	s	t	s
k	aa1	s	t	s

t	w	e	n	t	y
t	w	eh1	n	t	iy0

db	o	l l	a r	s
d	aa1	l	er0	z

t h	r	e e
th	r	iy1

o	n	e
w ah1	n	

Spelling / Sound Correspondence

a	n	d
ae1	n	d

t	w	e	n	t	y
t	w	eh1	n	t	iy0

th	r	ee
th	r	iy1

c	o	s	t	s
k	aa1	s	t	s

d	o	ll	ar	s
d	aa1	l	er0	z

o	n	e
w ah1	n	

Spelling / Sound Correspondence

a	n	d
ae1	n	d

t	w	e	n	t	y
t	w	eh1	n	t	iy0

t h	r	e e
th	r	iy1

c	o	s	t	s
k	aa1	s	t	s

d	o	l l	a r	s
d	aa1	l	er0	z

o	n	e
w ah1	n	

Spelling / Sound Correspondence

a	n	d
ae1	n	d

t	w	**e**	n	t	y
t	w	**eh1**	n	t	iy0

th	r	**ee**
th	r	**iy1**

c	o	s	t	s
k	aa1	s	t	s

d	o	ll	ar	s
d	aa1	l	er0	z

o	n	**e**
w ah1	n	

Spelling / Sound Correspondence Types

Modes of pronunciation:

t	w	e	n	t	y
t	w	eh1	n	t	iy0

x	y	z
eh1 k s	w ay1	z eh1 d z iy1

x	b	o	x
eh1 k s	b	aa1	k s

Spelling / Sound Correspondence Types

Clusters of correspondences:

c h	a	r	l	**e**	s
ch	aa0	r	l		z

c h	a	r	l	**e**	n	**e**
sh	aa0	r	l	**iy1**	n	

c h	i	a	n	**e**	s	**e**
k	iy0	aa0	n	**ey1**	z	**iy0**

Building pronunciation lexicons

- Ask human experts to annotate thousands of examples
  - Want to cover all of the words in the recorded prompts
  - Coverage for frequent words in the language
  - Coverage for “difficult” words
- Need for quality assurance:
  - Humans make mistakes
  - Especially on repetitive, monotonous tasks
  - Annotation guidelines may be unclear
  - Phenomena may be insufficiently understood

Need to continuously check human annotated data against a model.

Model checking examples

c	r	i	c k	e	t	e	r
k	r	ih1	k	ax	t		

c	r	i	c k	e	t	
k	r	ih1	k	ax	t	er0

r	i	p	e
t	ay1	p	

Data vs. model

c	r	i	c k	e	t	e	r
k	r	ih1	k	ax	t		

c	r	i	c k	e	t	
k	r	ih1	k	ax	t	er0

r	i	p	e
t	ay1	p	

c	c k	e	e
k	k	ax	

e r	i	i	r
er0	ih1	ay1	r

p	t
p	t

Alignment model checking for quality assurance

Simple (monotonic) alignment model.

Parameters: Pairs of (letters, phonemes), a/k/a “graphones”.

Simplest possible check: Does a proposed transcription of a word align with its orthography?

If yes: So far, so good. (Like any test, model checks mostly reveal defects.)

If no: There is a problem in the model or in the data. Revise, refine, repeat.

Data vs. model: A 19th century example

r	ou	gh
r	ah1	f

n	a	ti	o	n
n	ey1	sh	ax	n

w	o	m	e	n
w	ih1	m	ax	n

a	e	gh	m
ax	ey1	f	m

n	o	o	ou
n	ax	ih1	ah

r	ti	w
r	sh	w

Data vs. model: A 19th century example

r	o u	g h
r	ah1	f

n	a	t i	o	n
n	ey1	sh	ax	n

w	o	m	e	n
w	ih1	m	ax	n

g h	o	t i
f	ih1	sh

a	e	g h	m
ax	ey1	f	m

n	o	o	o u
n	ax	ih1	ah

r	t i	w
r	sh	w

Pronunciation models with latent alignments

Want to find the pronunciation of a previously unseen word.

Assume that all (orthography, pronunciation) pairs are generated as follows:

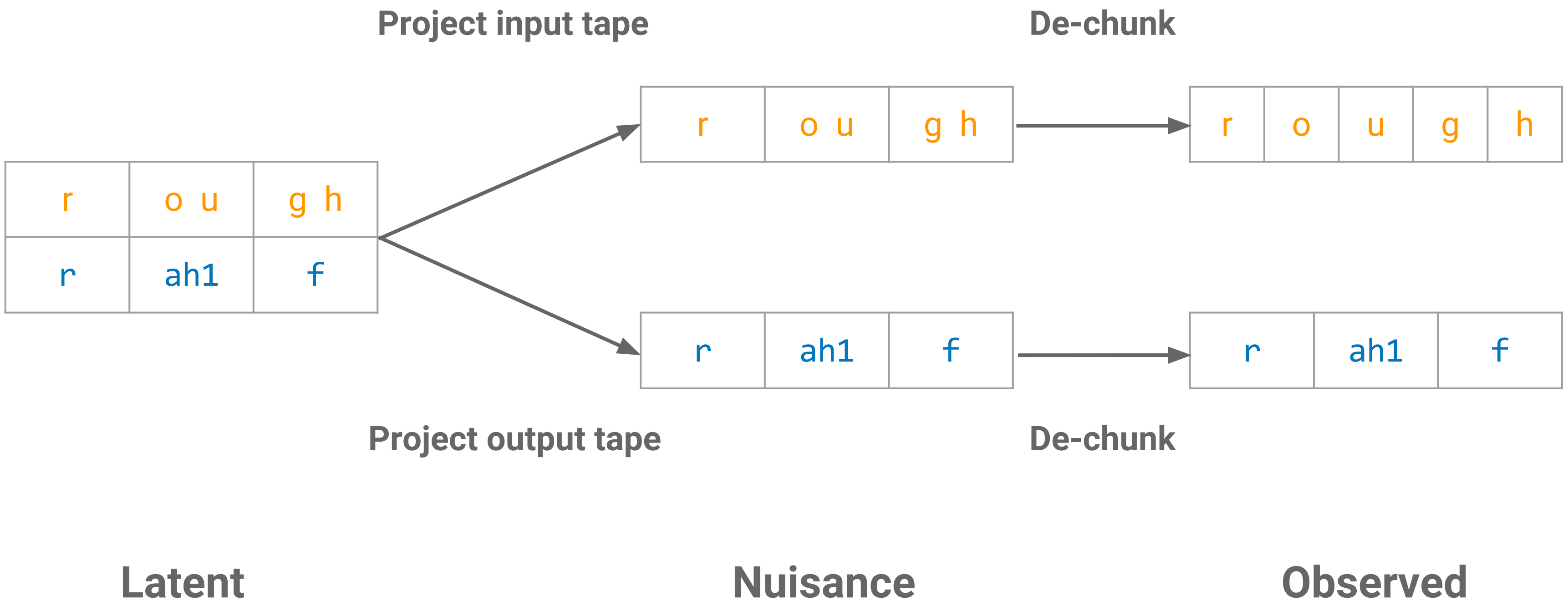
- Some process generates a latent alignment
- We observe the orthography and pronunciation as projections of the alignment.

Latent Monotonic Alignment Model (similar to SMT, but monotonic)

Some assumptions about the underlying generative process:

- Markov chain of order *n* (a hyperparameter)
- Graphone inventory (alphabet) is known ← **this is new and crucial**

Graphical model illustration



Project input tape

De-chunk

Project output tape

De-chunk

Latent

r	o u	g h
r	ah1	f

Nuisance

r	o u	g h

r	ah1	f

Observed

r	o	u	g	h

r	ah1	f

Simple LMAM estimation/training

Inputs:

- Examples of (orthography, pronunciation) pairs, i.e. pronunciation dictionary
- Graphone inventory

Output: Point estimate of model factors

Procedure:

- Align pronunciation dictionary with the given graphone inventory:
  - Imputes unobserved alignment
  - Potential issues around uniqueness of alignments
- Train *n*-gram language model on aligned data

LMAM predictive inference

Inputs: Estimated model components; List of orthographic words
Output: Pronunciations for input words

Standard inference in a (simple chain) graphical model:

- Inverse-project orthography into alignment lattice
- Score alignment lattice with *n*-gram model
- Project alignment lattice into phoneme lattice
- Decode, e.g. *k*-best paths

All steps can be expressed with standard operations on WFSTs

LMAM properties

- Conceptually extremely simple, yet highly competitive baseline
- Very compact models:
  - Store model in factorized form ← **this is new and crucial**
  - Use LOUDS compression to store *n*-gram model ← **this is new and crucial**
  - Size mostly a function of graphone inventory and hyperparameter *n*
  - Footprint <500kB achievable, depending on the language/data
- Efficient inference:
  - Choose graphones to give rise to acyclic lattices ← **this is new and crucial**
  - If all intermediate lattices are acyclic, all FST algorithms are linear in the FST size (V+E)
- Simple training:
  - Re-useable aligner (also used for quality assurance / model checking)
  - Off-the-shelf *n*-gram language model training (OpenGRM)

Building pronunciation dictionaries

- Define phoneme inventory for the language
- Define alignable graphones
- Repeat in batches:
  - Annotate data and continuously monitor alignments
  - Refine graphone inventory as needed
  - Impute alignments (review initially) and train pronunciation model
  - Model predicts pronunciations of unseen words, human experts correct them

Phonemic Analysis revisited

- Human factors
  - Model checking for quality control
- What if the word is not in the dictionary?
  - Use a pronunciation model to predict its pronunciation
- Pronunciation of a new word can be guessed from related words, but may differ
  - Post-lexical process: internal sandhi
- Phonological interactions across word boundaries
  - Post-lexical process: external sandhi

Phonemic post-processing, a/k/a post-lexical processing

Naive procedure: Find the pronunciation of each word independently, concatenate the word pronunciations to yield the pronunciation of the utterance.

(Similar in ASR: decoder graph is often $C \circ L^* \circ G$, with L a pronunciation lexicon.)

This does not work in French (*liaison*), Italian (*radoppiamento*), etc.

French liaison example

le**s m**isérables	`l e . m i . z e . R a b . l @`
le**s a**mis	`l e .` **`z`** `a . m i`
le**s h**ommes	`l e .` **`z`** `O m`
le**s h**éros	`l e . e . R o`

French liaison example

le**s m**isérables	l e . m i . z e . R a b . l @
le**s a**mis	l e . **z** a . m i
le**s h**ommes	l e . **z** O m
le**s h**éros	l e . e . R o

Typically solved by using special symbols in the phoneme inventory:

le**s**	l e (z)
héros	* e . R o (z)
hommes	O m (z)

French liaison example

les	`l e (z)`
héros	`* e . R o (z)`

The output of the lexicon is rewritten along the following lines:

```
(z) → ε / _ (*|k|g|t|d|n|p|b|m|...)
(z) → z / _ (i|y|u|e|2|o|E|9|O|a|...)
* → ε
```

This can be accomplished with a 3-state FST, e.g. compiled from a Thrax grammar.

Moral: Phonemes (lexicon output) can be whatever you need them to be.

(This is Basic Phonology.)

Orthographic pre-processing

Depending on the writing system, it may be useful (Bengali) or necessary (Korean) to deterministically rewrite the orthographic input string to a more convenient form.

“More convenient” roughly means “more segmentally phonemic”.

We want an orthographic form that is structurally well-matched with the phonological form. Ideal simplest case: one grapheme per phoneme and vice versa.

Restructuring the orthographic input only makes sense if it can be done deterministically and without any additional information.

Moral: Graphemes (lexicon input) can be whatever you need them to be.

Orthographic pre-processing in Bengali

Spelling	অণুবীক্ষণ										
Visual clusters	অ	ণু		বী		ক্ষ				ণ	
Codepoints	A	NNA	-U	BA	-II	KA	HASANT	SSA		NNA	
Graphemes	a	ṇ	u	b	ii	k		ṣ	a	ṇ	a
Phonemes	o	n	u	b	i	k k^h			ɔ	n	

Restructure the orthography to make the vowel inherent in every consonant letter explicit when not suppressed and no explicit vowel is present. (Similarly in related scripts.)

Leads to more compact, less ambiguous graphone lattices; and better estimates.

Orthographic pre-processing in Bengali

Spelling	অণুবীক্ষণ										
Visual clusters	অ	ণু		বী		ক্ষ				ণ	
Codepoints	A	NNA	-U	BA	-II	KA	HASANT	SSA		NNA	
Graphemes	a	ṇ	u	b	ii	k		ṣ	a	ṇ	a
Phonemes	o	n	u	b	i	k k^h			ɔ	n	

NNA	NNA	NNA
n	n o	n ɔ

BA	BA	BA
b	b o	b ɔ

ṇ	b
n	b

a	a	a
	o	ɔ

Phonemic Analysis summary

- Bootstrap lexicons by iterative refinement of a pronunciation model, which is used to predict unseen data and validate annotated data.
- Ideally suited for parametric generative models.
  We use and recommend simple Latent Monotonic Alignment Models.
  Does not work with nonparametric models!
- Requires active engagement with model and data. No shortcut here.
- Software available at github.com/googlei18n/language-resources and on the Docker image for the Lab Session.

Phrasing and Prosody??

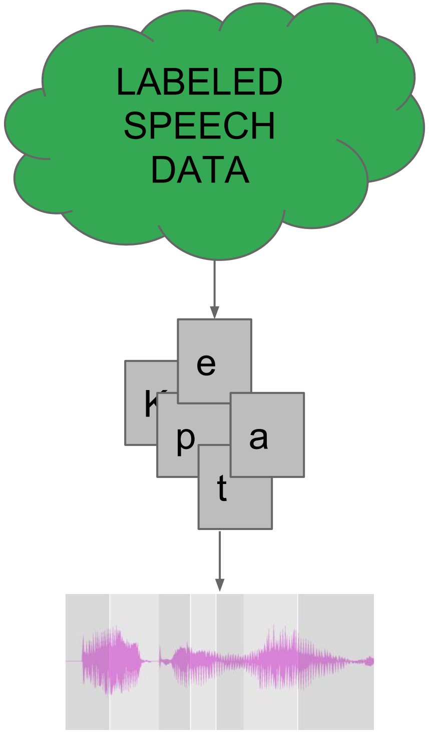
Phrasing and prosody are hard!

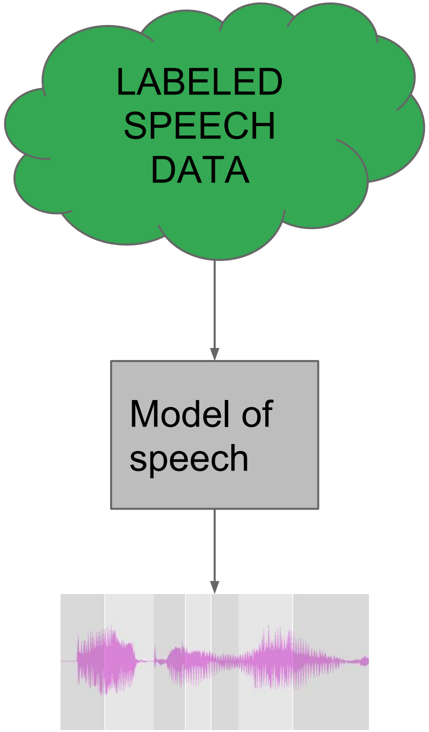
- Phrasing models based on pauses and/or punctuation are most common.
  - Rule based models available.
  - Statistical models can be trained.

- None of the explicit prosody models for Festival are very good
  - Unit selection often works okay if in-domain or close to in-domain.
  - Parametric is overly averaged, but at least consistent.

Synthesis

Unit Selection vs Parametric





Synthesis engines in Festival

Unit Selection:

Single Diphone	(mid 1990s)	Single instance of each diphone
CLUnits	(late 1990s)	Limited domain
Multisyn	(early 2000s)	Open domain

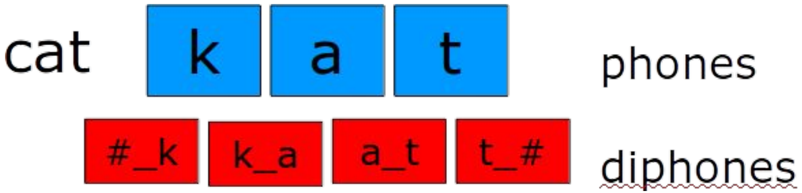
Parametric:

HTS	(early 2000s)	HTK based, HMM system
ClusterGen	(mid 2000s)	

Multisyn Unit Selection

Diphones are generally the type of units used

- A diphone is a chunk of speech which starts at the centre of one phone and ends at the centre of the next



How many diphones does a language have?

No. of phones squared?

Using a database for Unit Selection

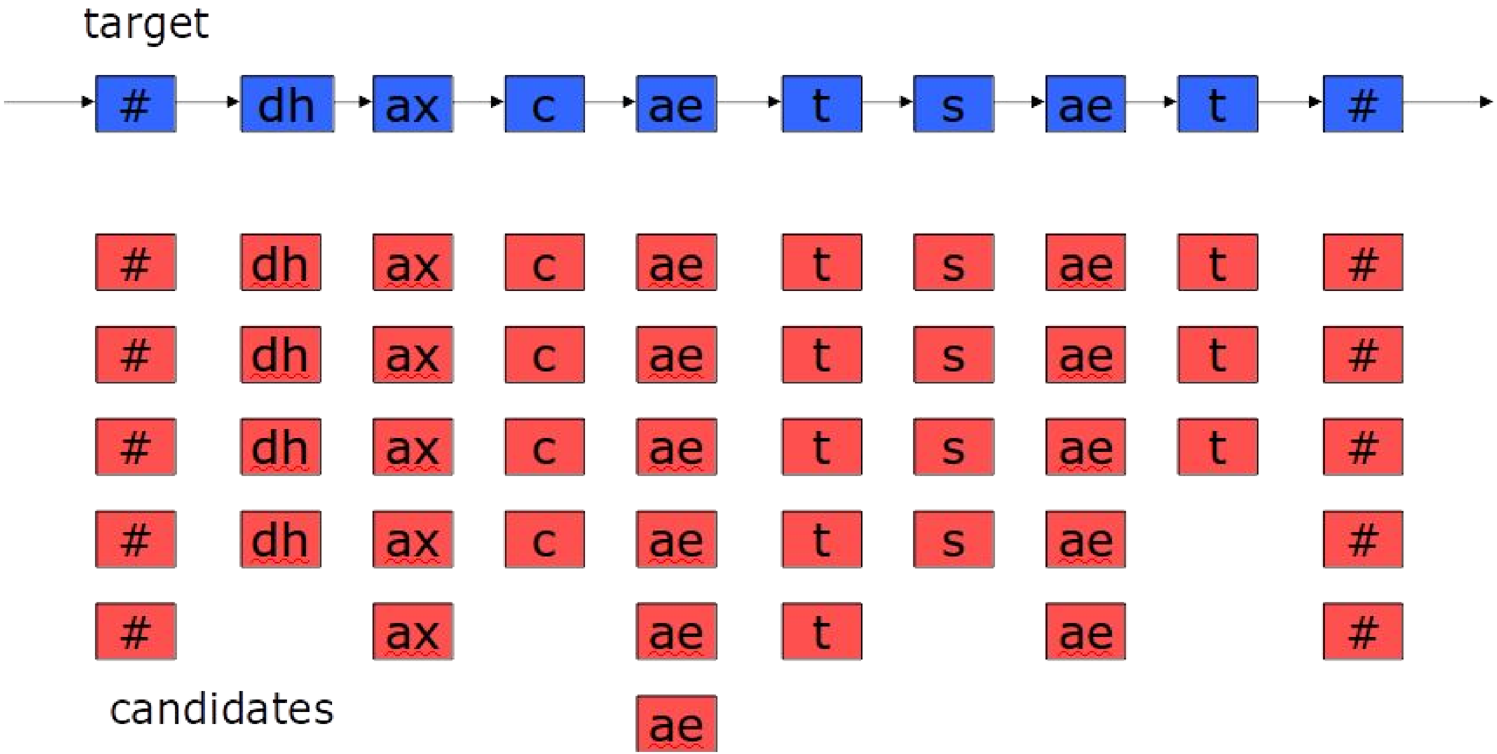
At synthesis time we are give a text and we are required to find the most suitable phone sequence from our database to concatenate together to produce suitable speech

First we perform linguistic analysis to come up with a description of the sounds and their contexts that we require

The result is a suitable linguistic structure

(which is similar to the structure our database is annotated with).

Unit selection speech synthesis



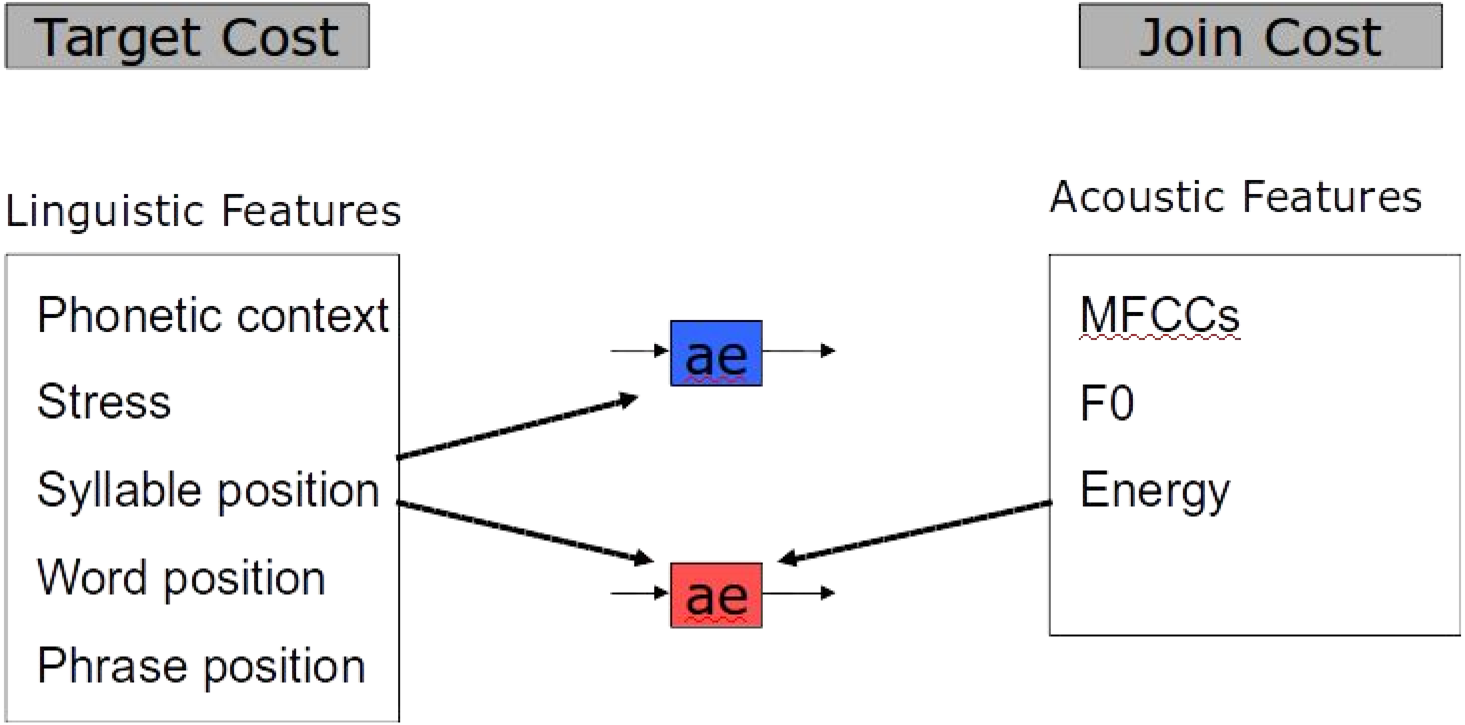
target

→ dh → ax → c → ae → t → s → ae → t →

#	dh	ax	c	ae	t	s	ae	t	#
#	dh	ax	c	ae	t	s	ae	t	#
#	dh	ax	c	ae	t	s	ae	t	#
#	dh	ax	c	ae	t	s	ae		#
#		ax		ae	t		ae		#
				ae					

candidates

How do we do this?



Target Cost

Join Cost

Linguistic Features

- Phonetic context
- Stress
- Syllable position
- Word position
- Phrase position

ae

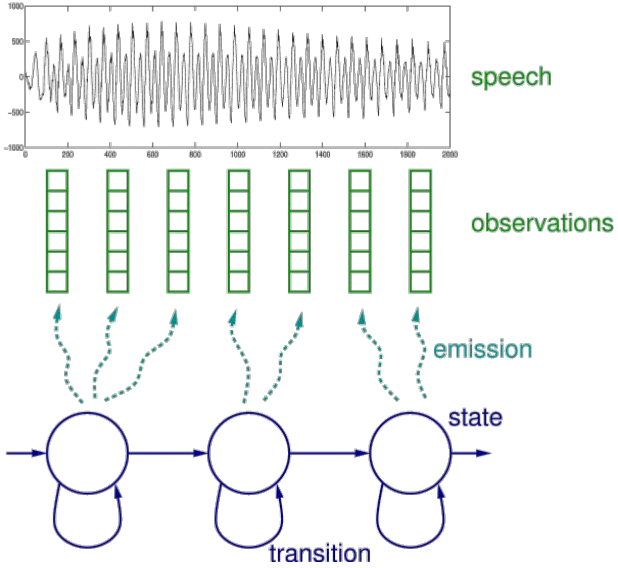
ae

Acoustic Features

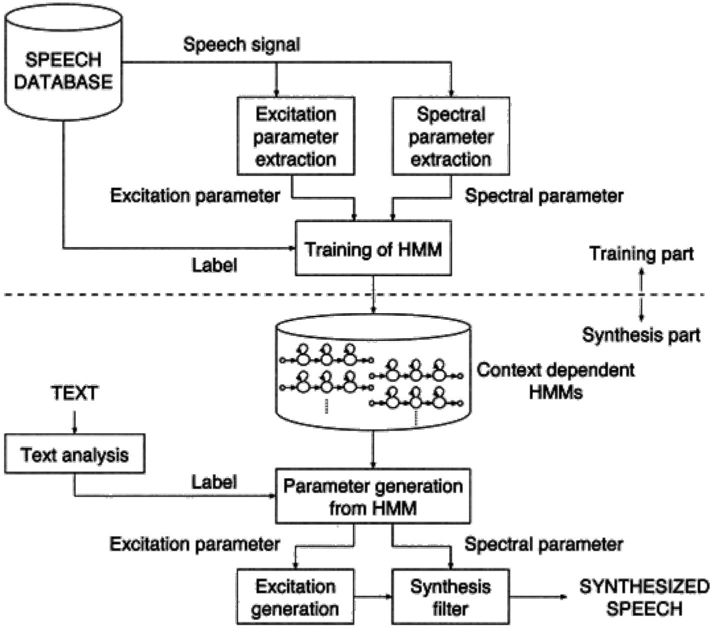
- MFCCs
- F0
- Energy

HTS based parametric

We use Hidden Markov models (HMMs) for speech recognition. Can we use them for speech synthesis?

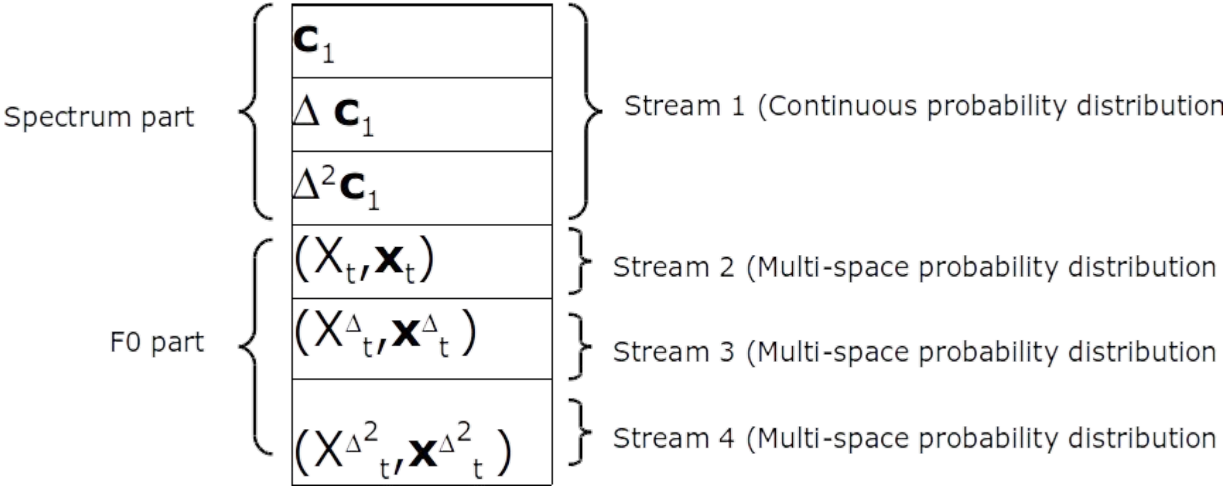


HMM-based speech synthesis



from: An HMM-based approach to multilingual speech synthesis, Tokuda, Zen & Black, in Text to speech synthesis: New paradigms and advances; Prentice Hall: New Jersey, 2004

HMM output vectors



Spectrum part

$\mathbf{c}_1$

$\Delta\ \mathbf{c}_1$

$\Delta^2\mathbf{c}_1$

Stream 1 (Continuous probability distribution

F0 part

$(X_t, \mathbf{x}_t)$

$(X^{\Delta}_t, \mathbf{x}^{\Delta}_t)$

$(X^{\Delta^2}_t, \mathbf{x}^{\Delta^2}_t)$

Stream 2 (Multi-space probability distribution

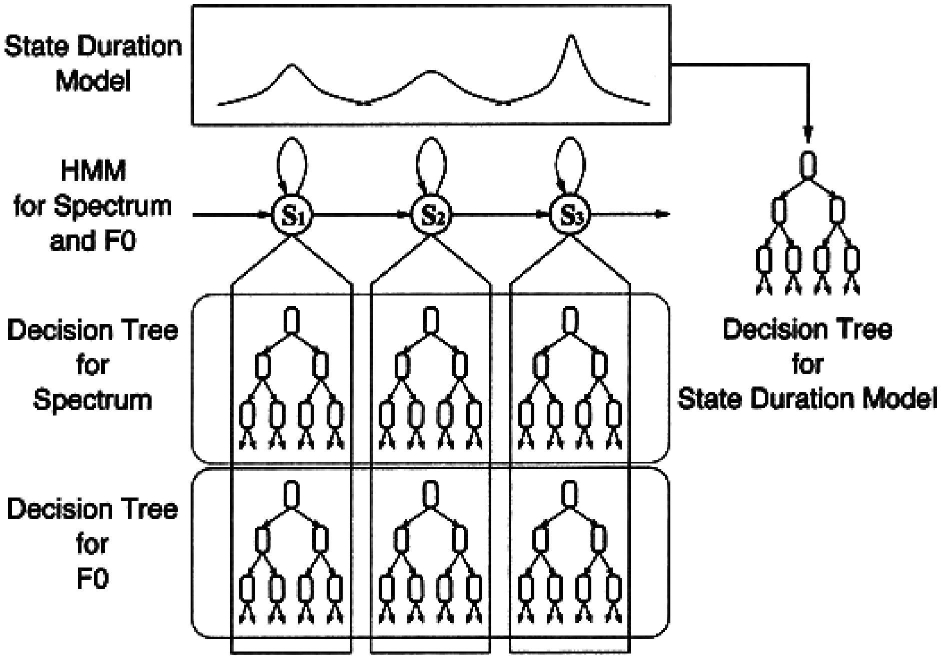
Stream 3 (Multi-space probability distribution

Stream 4 (Multi-space probability distribution

Models additionally have state duration densities to model duration

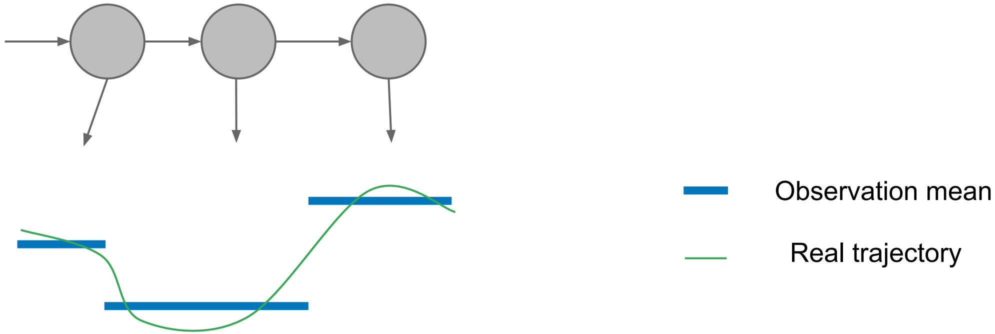
Decision tree based context clustering

As with unit selection difficult to record all the phonetic/linguistic contexts you need. Can use decision trees to cluster



Synthesis

- Compute phone sequence as we would for any other synthesis.
- Concatenate the trained models for each phone.
- Generate observation vectors, using MLPG algorithm
- Convert spectral duration and pitch information into a speech waveform.



Clustergen parametric

Similar to HTS but generates 1 observation per frame, rather than 1 per state.

- Per frame observations vary, so MLPG is not required, and simple smoothing can be performed instead
- ...or trajectory modelling can be performed more explicitly in various ways

An introduction to scheme

An introduction to scheme

What is scheme?

- An interpreted shell language
- A variant of LISP

Why scheme?

- Allows maximum flexibility
    - Rapid prototyping
    - Flexible configuration

Is completely embedded in Festival, no package dependencies.

The good the bad and the ugly

The downside is that scheme and LISP are a bit obscure, and not the easiest language to work with.

- The brackets will drive you mad!!!

… So we provide a gentle introduction here. This is intended to be enough information (often simplified) to enable the use and understanding of scheme in Festival rather than anything else.

Scheme for beginners

here is only one type of statement in scheme, it looks like this:

(function_name arg1 arg2 …)

And the only thing scheme does is to take expressions like the one above and evaluate them, replacing the statement with the result of the evaluation

Data structures in scheme

Simple atomic data types:

- t, nil
- 1,2,3,…

Creating new atomic data items:

```
 > (quote hi)
hi
```

The quote function is the only function to not evaluate its argument!

Short hand: ‘hi, ‘a, ‘b, ‘1, ‘2, ‘label, ‘strawberry

The point of atoms is that they evaluate to themselves.

Data structures in scheme

Strings:

“hi”, “label”, “strawberry”

Strings are not atoms!   “strawberry” ≠ ‘strawberry

Although many festival functions can take either as their arguments

Variables and function names

Names that are unquoted are assumed to be variables or function names.
Variables can be set with the command set!

```
> (set! v1 “hello”)
“hello”

> (set v1 (+ 1 2))
3

> v1
“hello”

> v2
3
```

Functions can be defined with the command define which we will hear more about later.

Complex data types

Lists are the main data type in scheme.
There are 2 ways to generate lists.

```
> (list 1 2 3 ‘a ‘b ‘c)
(1 2 3 a b c)

> `(1 2 3 a b c)
(1 2 3 a b c)

> (list 1 2 3 `(a b c))
(1 2 3 (a b c))

> (list 1 2 3 v1 v2)
(1 2 3 1 3)
```

Processing lists

Two list accessing function car and cdr

- car – return the first item in the list
- cdr – return the tail of the list

```
> (set l1  ‘(1 (“hello” “world”) 2))
> (car l1)
1

> (cdr l1)
((“hello” “world”) 2)

> (car (cdr l1))
(“hello” “world”)
```

Defining functions

An example of a function definition:

```
(define (foo a1 a2 a3)
  (let ((v1 (+a1 a2))
        (v2 nil))
    (set! v2 (+ v1 a3))))
```

```
> (foo 1 2 3)
6

> (foo 1 2)
SIOD ERROR: too few arguments : …
```

More on let

```
(let ((v1 val1) (v2 val2) v3 v4)
   BODY)
```

Let defines the scope of local variables

- It take 2 arguments a list of variables and a body of code
  - The variables are defined for the duration of the body
  - The variables can either be lists: where a value is specified
  - or just a variable name

Testing equality

(eq? a b)	true if the same object
(equal? a b)	true if recursively equal
(string-equal a b)	true if strings match
(string-matches a b)	true if regex matches
(< a b)	true if a < b
(> a b)	true if a > b

Some scheme programming constructs…

(cond (test1 do1) (test2 do2) …)

```
(cond
     ((not (number ?))
      (print “x is not a number”))
     ((< 3 x)
      (print “x is < 3”))
     (t
     (print “ x is >= 3”)))
```

Some scheme programming constructs…

(mapcar (lambda (X) DO_EACH_X) LIST_OF_Xs)

```
(mapcar
  (lambda (x)
    (* 2 x))
  `(1 2 3 4 5))
```

```
(2 4 6 8 10)
```

To keep the procedural programmers happy…

(while CONDITION BODY)

```
(while (> x 0)
   (print x)
   (set! x (- x 1)))
```

(if CONDITION TRUE_DO FALSE_DO)

```
(if (eq? x 1) (print “x is 1”) (print “x is not 1”))
```

Loading files

(load “path/filename.scm”)

Loads the scheme file and evaluates its contents

Useful as long expressions and function definitions are difficult to get right on the command line.

Scheme in Festival

Scheme is used by Festival in a number of ways:

- To control the flow of the synthesis process
- To prototype new methods
- To allow easy access to the data structures as synthesis before, during and after synthesis.

Getting help

Most scheme functions have documentation built in. Type the name of the function and press <ESC> followed by H

```
> (SayText<ESC>H
(SayText TEXT)
TEXT, a string, is rendered as speech.
```

Also, pressing TAB halfway through a function name will give you a list of possible completions

Failing that, the festival manual is online at:

http://www.cstr.ed.ac.uk/projects/festival/manual

Festival internal data structures

Data structures in Festival

The top level data object will we be most interested in is the utterance.

- Utterances store data as a heterogeneous relation graph (HRG)
- The utterance is the container object in which speech synthesis occurs.
- Information is added to the utterance by each stage of the synthesis process.

Most of this information is stored in HRGs as relations, items and features

Items and features

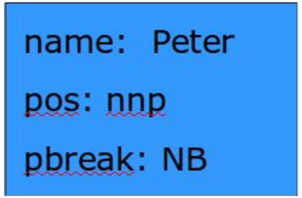
Each chunk of data is stored as an item.

- phones, words, syllables, pitch accents,...

Each item is described by a set of features.

- name, end, ...
- e.g. for a word: pos
- e.g. for a syllable: stress

A word item

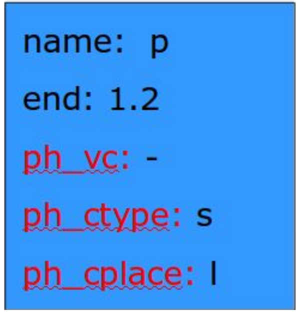


name: Peter

pos: nnp

pbreak: NB

A segment item



name: p

end: 1.2

ph_vc: -

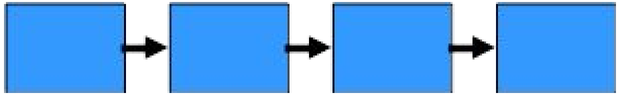
ph_ctype: s

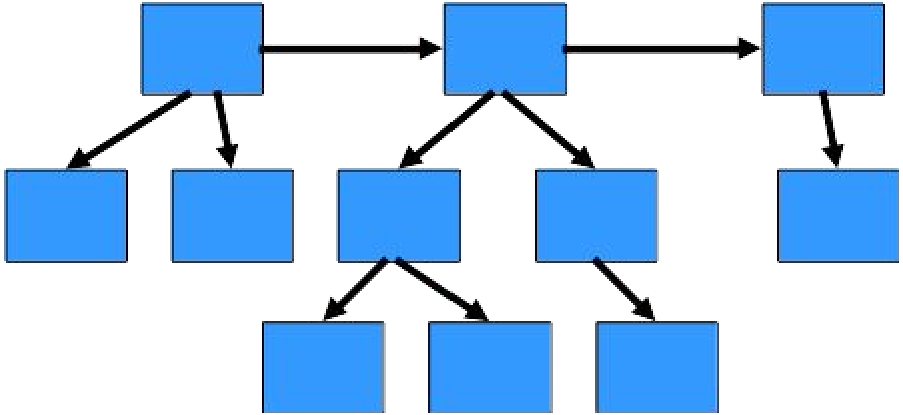
ph_cplace: l

Items and Relations

Items don’t exist on their own, but each item is part of one or more relation.

Relations come in 2 flavours, *lists* and *lists of trees*.





Some standard relations used by Festival

Token – pre-processed input tokens

Word – actual words (e.g. eighty four)

Phrase – phrases

Syllable – syllables

Segment – phones

SylStructure* – syllabic structure

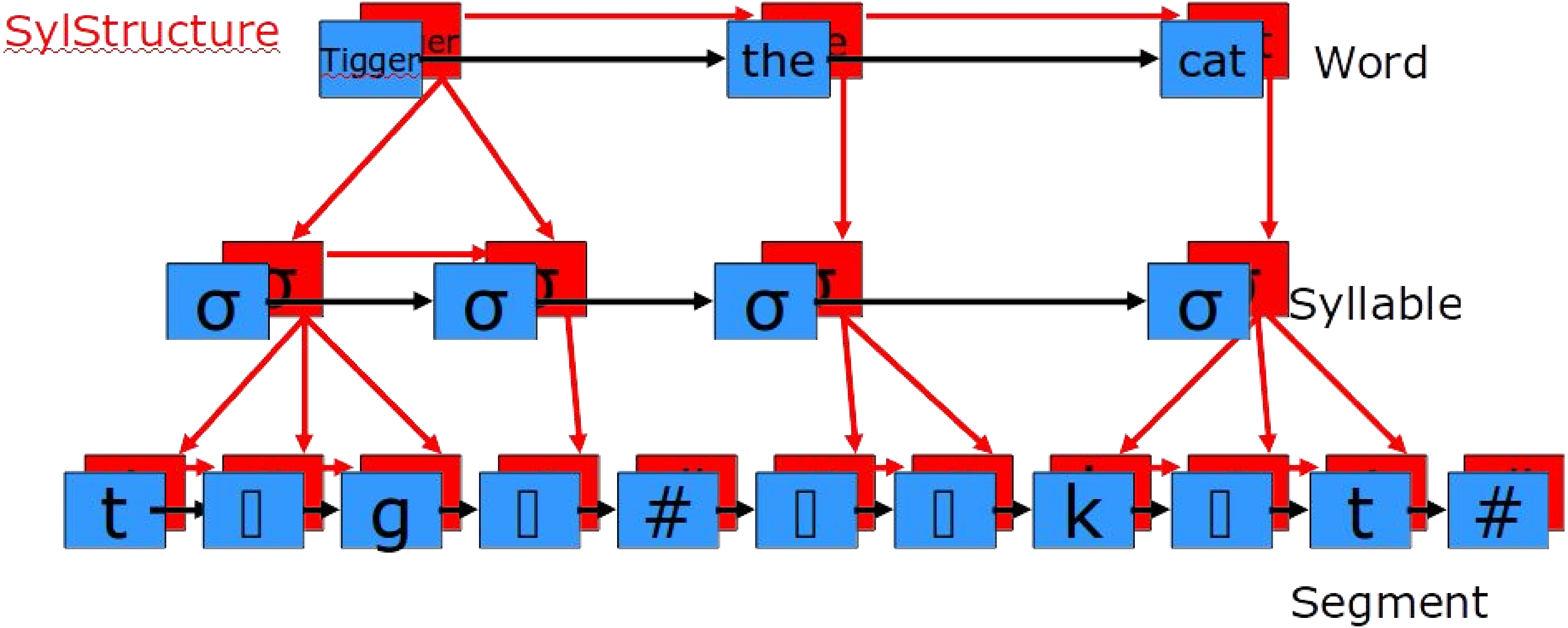
IntEvent – intonation events

Intonation* – intonation structure

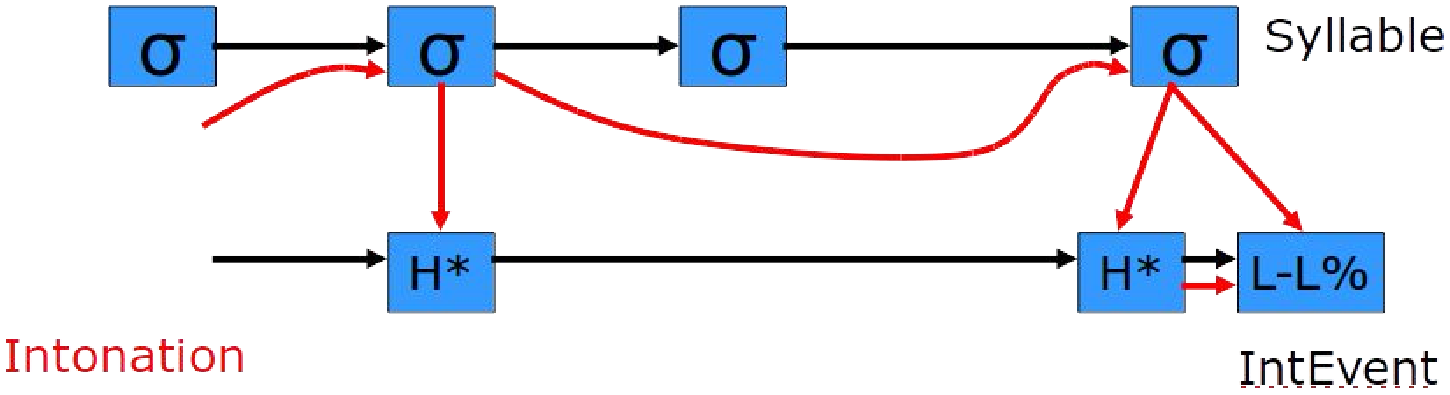
Unit – chosen unit sequence

* – list of tree relations

When items are in more than one relation



When items are in more than one relation



So a syllable item can be in up to 3 relations:

- Syllable, Sylstructure and Intonation

Accessing utterances in scheme

```
> (set! utt (Utterance Text “Hello world.”))
#<Utterance 0xa1843a0>

> (utt.synth utt)
…
#<Utterance 0xa1843a0>

> (utt.relationnames utt)
(Token Word Phrase …)

> (set! segs (utt.relation.items utt ‘Segment))
(#<item 0xb261a08> #<item 0xb247808> #<item 0xb2609d0>...)
```

Accessing utterances in scheme

```
> (set! seg1 (car segs))
#<item 0xb261a08>

>(set! seg2 (car (cdr segs)))
#<item 0xb247808>

> (item.feat seg1 “name”)
“pau”

> (item.feat seg2 “name”)
“hh”
```

Accessing utterances in scheme

```
> (utt.relation.print utt ‘Segment)
()
id _17 ; name pau ; dur_factor 0 ; end 0.22 ; source_end 0.081826 ;
id _7 ; name hh ; dur_factor -0.296956 ; end 0.277954 ; source_end 0.188655 ;
id _8 ; name ax ; dur_factor -0.317324 ; end 0.320176 ; source_end 0.289519 ;
id _9 ; name l ; dur_factor 0.240634 ; end 0.399659 ; source_end 0.378457 ;
id _11 ; name ow ; dur_factor 0.0696307 ; end 0.550046 ; source_end 0.550021 ;
id _13 ; name w ; dur_factor 0.636568 ; end 0.625551 ; source_end 0.690708 ;
id _14 ; name er ; dur_factor 0.520952 ; end 0.725881 ; source_end 0.800834 ;
id _15 ; name l ; dur_factor 0.520952 ; end 0.813381 ; source_end 0.912022 ;
id _16 ; name d ; dur_factor 0.730381 ; end 0.883052 ; source_end 1.09058 ;
id _18 ; name pau ; dur_factor 0 ; end 1.10305 ; source_end 1.37287 ;
Nil
```

Moving around a relation

(item.next ITEM)
(item.prev ITEM)
(item.parent ITEM)
(item.daughter1 ITEM)
(item.daughter2 ITEM)

```
> (item.feat (item.next seg2) “name”))
“ax”

> (item.feat (item.next (item.next seg2)) “name”))
“l”
```

Moving between relations

Recall that an item can be in more than one relation.

- Any instance of an item is considered to be held with respect to a single relation at any time.
- The functions like item.next only allow you to move around that relation.

This is important so lets look at it again!

Each item can be in multiple relations

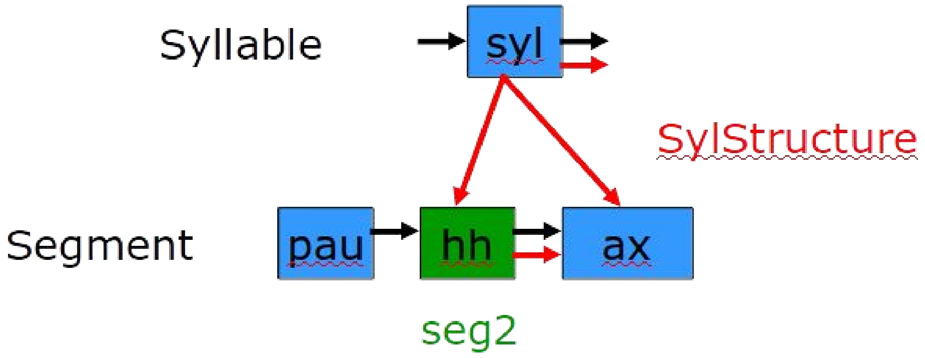
- In each relation each item has certain links to other items
    - Segment items link to other segments
    - Syllable items link to other syllables
- If you want to move from syllables to segments you need to reference with respect to the SylStructure relation
    - The parent and daughter links are only available in this relation

Moving between relations

We can change the relation which an item is being held in reference to:
(item.relation ITEM RELATIONNAME)

And there are some shortcuts for moving around:
(item.relation.next ITEM RELATIONNAME)
(item.relation.prev ITEM RELATIONNAME)
(item.relation.parent ITEM RELATIONNAME)
(item.relation.daughter1 ITEM RELATIONNAME)
(item.relation.daughtern ITEM RELATIONNAME)

Moving between relations

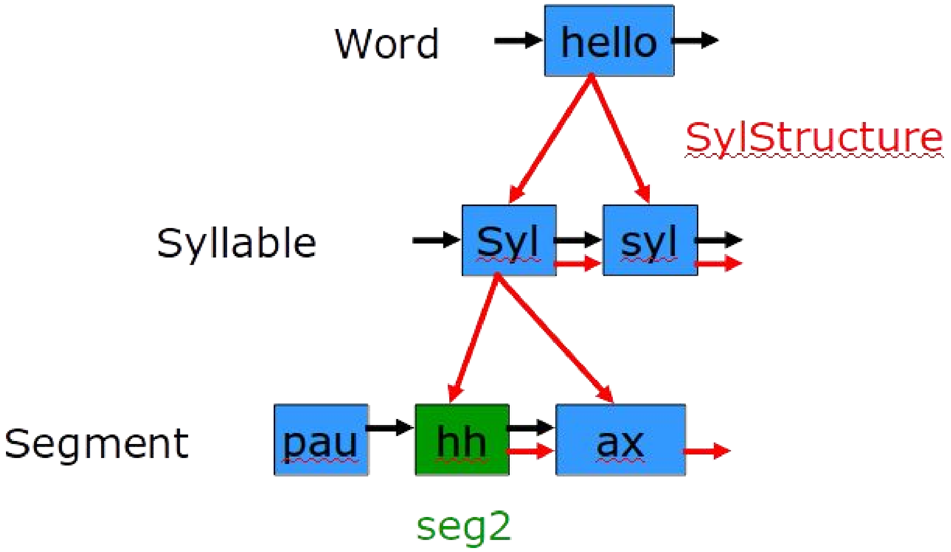


[Recall seg2 is a Segment item]

```
> (item.feat (item.parent seg2) “name”)
nil
```

```
> (item.feat (item.relation.parent seg2 ‘SylStructure) “name”)
“syl”
```

Moving between relations



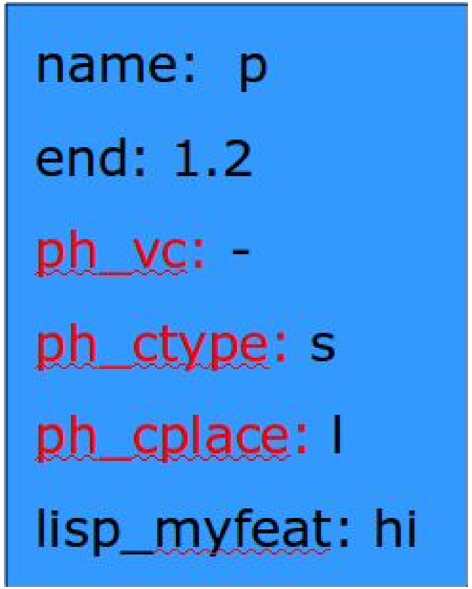
```
> (item.feat (item.parent (item.relation.parent seg2 'SylStructure)) "name")
"hello"
```

More on features

Features come in a number of types:

- Real features
    - Physical data in the item
- Feature functions
    - A predefined function
- User defined feature functions
    - A user defined function

A segment item



name: p

end: 1.2

ph_vc: -

ph_ctype: s

ph_cplace: l

lisp_myfeat: hi

Feature paths

Sometimes it is possible to move around a relation using special path directives in a feature name

```
(item.feat seg “R:SylStructure.parent.parent.name”)
```

Feature paths

R:<relationname>.	ref. wrt. this relation Parent.
daughter1.	first daughter
daughtern.	last daughter
n.	next
p.	previous
nn.	next next
pp.	previous previous
lisp_<functionname>	user defined function

Lab Session

Docker image for this tutorial

On your host machine:

```
$ docker pull mjansche/tts-tutorial-sltu2016

$ docker run --rm -it mjansche/tts-tutorial-sltu2016

root@container:/usr/local/src# ls

af_classifier  festvox              goog_af_unison_wav_22k  textnorm_exercise
af_verbalizer  g2p                  speech_tools
festival       goog_af_unison_text  test_sparrowhawk.scm

root@container:/usr/local/src#
```

Inside the container

GNU/Linux OS, latest Ubuntu LTS release

Install any tools you might need:

```
# apt-get update
# apt-get install vim
```

Building a voice with FestVox

Building a Festival Clustergen voice with FestVox

```
/usr/local/src# mkdir voice_building

/usr/local/src# cd voice_building

/usr/local/src/voice_building# ../goog_af_unison_text/build-voice.sh
```

Exercise: Go through `build-voice.sh` step by step

Pronunciation models (G2P)

Using the Afrikaans pronunciation (G2P) model

```
/usr/local/src# cd g2p

/usr/local/src/g2p# make -C runtime
make: Entering directory '/usr/local/src/g2p/runtime'
g++ -std=c++11 -O2 -I/usr/local/include -L/usr/local/lib/fst -
L/usr/local/lib  g2p-lookup.cc  -lfstngram -lfst -ldl -o g2p-lookup

/usr/local/src/g2p# ./g2p.sh af
skeertuig
skeertuig  s k i@ r t 9y x   0.993945   0.993945
```

Using the Afrikaans pronunciation (G2P) model

```
/usr/local/src/g2p# ./g2p.sh af --v=1

b
INFO: 2 hypotheses searched
INFO: 1 pronunciation found
b   b  0.995447   0.995447

e
INFO: 5 hypotheses searched
INFO: 1 pronunciation found
e   @  0.590727   0.590727
```

Using the Afrikaans pronunciation (G2P) model

```
/usr/local/src/g2p# ./g2p.sh af --v=1 --max_prons=5 \
  --real_pruning_threshold=0
b
INFO: 2 pronunciations found
b    b    0.995447      0.995447
b    p    0.00455283    1

e
INFO: 5 pronunciations found
e    @    0.590727      0.590727
e    E    0.174339      0.765067
e    i    0.0908592     0.855926
e    i@   0.0894359     0.945362
e    {    0.0546376     1
```

Using the Afrikaans pronunciation (G2P) model

```
/usr/local/src/g2p# ./g2p.sh af --v=1

be
INFO: 10 hypotheses searched
INFO: 1 pronunciation found
be  b @  0.921322  0.921322

bebebebebebebebebe
INFO: 1e+09 hypotheses searched
INFO: 1 pronunciation found
bebebebebebebebebe  b @ b @ b @ b @ b @ b @ b @ b @ b @  0.531657  0.531657
```

Building a G2P model

```
/usr/local/src# cd g2p/models/af

# lexicon-diagnostics --alignables=graphone_alignables.txt \
  --filter --unique_alignments lex_regular.txt > lex_aligned.txt
PASS

# cut -f3 lex_aligned.txt | farcompilestrings --keep_symbols \
  --symbols=graphone.syms --generate_keys=6 > train.far
```

Now we have aligned data stored as a collection of FSAs in train.far.

Building a G2P model

```
# ngramcount --order=2 train.far |
  ngrammake --method=kneser_ney --backoff |
  ngramfinalize --phi_label=0 --to_runtime_model > model2.fst

# echo warmlugballon |
  g2p-lookup--bytes_to_graphones=bytes_to_graphones.fst \
  --phonemes_to_graphones=phonemes_to_graphones.fst \
  --graphone_model=model2.fst
warmlugballon  v a r @ m l 9 x b a l O n    0.166418   0.166418
warmlugballon  v A: r m l 9 x b a l O n     0.149261   0.315679
warmlugballon  v a r @ m l 9 x b a l u@ n   0.113459   0.429138
```

Building a G2P model

```
# ngramcount --order=6 train.far |
  ngrammake --method=kneser_ney --backoff |
  ngramfinalize --phi_label=0 --to_runtime_model > model3.fst

# echo warmlugballon |
  g2p-lookup--bytes_to_graphones=bytes_to_graphones.fst \
  --phonemes_to_graphones=phonemes_to_graphones.fst \
  --graphone_model=model2.fst
warmlugballon  v a r @ m l 9 x b a l O n    0.44165    0.44165
```

Text normalization

Working with Thrax text normalization grammars

```
/usr/local/src# cd textnorm_exercise

textnorm_exercise# thraxmakedep number_names.grm

textnorm_exercise# make
thraxcompiler --input_grammar=byte.grm --output_far=byte.far
[snip]

textnorm_exercise# ./tester.py number_names.far number_names.tsv
[snip]
All tests pass!!
```

Working with Thrax text normalization grammars

```
textnorm_exercise# thraxrewrite-tester --far=number_names.far \
  --rules=CARDINAL_NUMBER_NAME

Input string: 1
Output string: een

Input string: 2
Output string: twee

Input string: 3
Output string: drie
```

Thanks!

Special thanks to Pasindu De Silva, Linne Ha, and Oddur Kjartansson from Google/NEMO; and to Marelie Davel, Charl van Heerden, Daniel van Niekerk, and Niel Kleynhans from NWU.