Finite-State Technology
Torbjörn Lager,
Göteborg University
Part 2

In this talk:
- Determinization and minimization of FSAs
- Time and space matters
- Representing word lists
- Determinization and minimization of FSTs
- Representing lexica
- More about the regular expression calculus
- LEXC and TWOLC

Non-deterministic FSAs
- At least one state has more than one transition leading from it labelled with the same symbol

Determinization of FSAs
- Any non-deterministic FSA can be transformed into an equivalent deterministic FSA.
- Example:

Efficiency
- Time complexity of recognition of a string with a deterministic FSA:
  - linear (O(n)) in the length of the string
  - constant (O(1)) in the size of the automaton

Minimization of FSAs
- Any (deterministic) FSA can be transformed into an equivalent FSA that has a minimal number of states.
- Minimize for space!
Representing word lists

- Think of a word list as a regular language
- Use the calculus of regular expressions to query and update the wordlist
- Determinize for speed!
- Minimize for space!

Transducers and determinism

- For FSTs the question of determinism is more complex:

```
0 a 1
      
b x
```

Transducers and determinism

- Deterministic transducers
  - Have deterministic underlying automata
- Sequential transducers
  - Are deterministic one way (not necessarily the other)
- Subsequential transducers
  - Have additional output at final state
- P-subsequential transducers
  - Have P additional outputs at final states

Many transducers can be determinized in the sense that they can be turned into sequential transducers.
They thus become very efficient (O(n))!

Representing lexica

- Think of a lexicon as a regular relation (i.e. as a set of pairs of strings)
- Use the calculus of regular expressions to query and update the lexicon
- Determinize for speed!
- Minimize for space!

Lexical transducers

```
[leave %VB x. leave]  
[leave %VBZ x. leaves]  
[leave %VBD x. left]
```

compiles into

```
0 1 2 3
      
4 5 6 7
 7 0
```

FST - Torbjörn Lager, GU
Lexical transducers

\[
\{ \text{leave} \times \text{VB} \times \text{leave} \} \\
\{ \text{leave} \times \text{VBZ} \times \text{leave} \} \\
\{ \text{leave} \times \text{VBD} \times \text{left} \} \\
\]

denotes
compiles into
generates
accepts

Lexical lookup (and lookdown)

- \text{xfst}[1]: apply up leaves
- \text{leave+VBZ}

- \text{xfst}[1]: apply down \text{leave+VBZ}
- \text{leaves}

XFST commands for working with word lists and lexica

- read text
- read spaced-text
- write text
- write spaced-text
- apply down
- apply up

- print words
- print lower-words
- print upper-words
- lower-side net
- upper-side net

Lexicon representation and lookup: issues and problems

- Issues
- Compounds
- Idioms
- Dates, etc.
- Unknown words
- Potential problems
- Transducer blowup

More about the regular expression calculus

- Composition
- Extraction
- Application = composition + extraction

Composition examples

- \text{XFST operator: .o.}
- Examples:
  \[
  \{ a|b|c \}^* \Delta \{ \neg[a \wedge a] \Delta \neg[b \wedge b] \Delta \neg[c \wedge c] \}
  \]
  \[
  \{ \neg[A \wedge A] \Delta \neg[B \wedge B] \Delta \neg[C \wedge C] \}
  \Delta
  \{A:a|B:b|C:c\}^*
  \]
Extraction

- XFST extraction operators: .1 and .2
- Example

```plaintext
[[leave %VB x. leave] |
[leave %VBZ x. leaves] |
[leave %VBD x. left]] .2
```

denotes

("leave" "leaves" "left")

Application = composition + extraction

- Simulating apply down with .o. and .2:

```plaintext
xfst[0]: regex a -> x ;
xfst[1]: apply down abc
xbc
xfst[0]: regex [a b c o. a -> x] .2 ;
xfst[1]: print words
xbc
```

More about lexica

- By just enumerating the lemma - surface form relation, we miss lots of phonological and morphological generalisations
- Instead, we may choose to overgenerate on (e.g.) the surface level, and write rules (and compile them into transducers) which fix that
- This, of course, is what Two-Level Morphology is all about!
- But note that the resulting transducer is the same in both cases!

Application = composition + extraction

- Generalizing apply down with .o. and .2:

```plaintext
xfst[0]: regex [a b [c|d] o. a -> x] .2 ;
xfst[1]: print words
xbc
xbd
```

Example: looking up "leaves" in XFST

```plaintext
xfst[0]: define Lex [[leave %VB x. leave] |
[leave %VBZ x. leaves] |
[leave %VBD x. left]] ;
xfst[0]: regex [Lex o. [leaves]] .1
xfst[1]: print words
leaveVBZ
```

Xerox tools for lexicon construction

- LEXC
  - Lexicon compiler - for building finite-state lexica. Syntax and algorithms optimized for this task. Produces transducers in standard XEROX binary format. Rules (XFST or TWOLC) may be imported.
- TWOLC
  - Compiler for two-level rules (Koskennemi 1983)