

# Finite State Morphology Tutorial

Miriam Butt and Tina Bögel

Konstanz

CLT 09, Lahore

# Content

The tutorial will be split into two parts:

# Content

The tutorial will be split into two parts:

- Theory of Finite State Morphology

# Content

The tutorial will be split into two parts:

- Theory of Finite State Morphology
  - some facts — the book & the software

# Content

The tutorial will be split into two parts:

- Theory of Finite State Morphology
  - some facts — the book & the software
  - some basic knowledge — finite state morphology

# Content

The tutorial will be split into two parts:

- Theory of Finite State Morphology
  - some facts — the book & the software
  - some basic knowledge — finite state morphology
  - building networks with xfst

# Content

The tutorial will be split into two parts:

- Theory of Finite State Morphology
  - some facts — the book & the software
  - some basic knowledge — finite state morphology
  - building networks with xfst
    - The Lexicon

# Content

The tutorial will be split into two parts:

- Theory of Finite State Morphology
  - some facts — the book & the software
  - some basic knowledge — finite state morphology
  - building networks with xfst
    - The Lexicon
    - Regular Expressions



# Content

The tutorial will be split into two parts:

- Theory of Finite State Morphology
  - some facts — the book & the software
  - some basic knowledge — finite state morphology
  - building networks with xfst
    - The Lexicon
    - Regular Expressions
    - The Interface

# Content

The tutorial will be split into two parts:

- Theory of Finite State Morphology
  - some facts — the book & the software
  - some basic knowledge — finite state morphology
  - building networks with xfst
    - The Lexicon
    - Regular Expressions
    - The Interface
  - possible applications

# Content

The tutorial will be split into two parts:

- Theory of Finite State Morphology
  - some facts — the book & the software
  - some basic knowledge — finite state morphology
  - building networks with xfst
    - The Lexicon
    - Regular Expressions
    - The Interface
  - possible applications
- Practical application of Finite State Morphology

# Finite State Morphology - The Book

- Lauri Karttunen and Kenneth R. Beesley (2003)

# Finite State Morphology - The Book

- Lauri Karttunen and Kenneth R. Beesley (2003)
- Xerox finite-state tools and techniques for morphological analysis and generation

# Finite State Morphology - The Book

- Lauri Karttunen and Kenneth R. Beesley (2003)
- Xerox finite-state tools and techniques for morphological analysis and generation
  - **lexc** → high-level language for specifying lexicons

# Finite State Morphology - The Book

- Lauri Karttunen and Kenneth R. Beesley (2003)
- Xerox finite-state tools and techniques for morphological analysis and generation
  - **lexc** → high-level language for specifying lexicons
  - **xfst** → a) interface providing regular-expression compiler  
b) access to the Xerox Finite State Calculus

# Finite State Morphology - The Book

- Lauri Karttunen and Kenneth R. Beesley (2003)
- Xerox finite-state tools and techniques for morphological analysis and generation
  - **lexc** → high-level language for specifying lexicons
  - **xfst** → a) interface providing regular-expression compiler  
b) access to the Xerox Finite State Calculus
  - runtime applications **tokenize** and **lookup**



# A Short Look at the Xfst Interface

`xfst[3]:`

The xfst Interface is a command line

# A Short Look at the Xfst Interface

`xfst[3]:`

The xfst Interface is a command line

- 1 open and process files

# A Short Look at the Xfst Interface

`xfst[3]:`

The xfst Interface is a command line

- 1 open and process files
- 2 enter commands/REGULAR EXPRESSIONS directly

# A Short Look at the Xfst Interface

```
xfst[3]:
```

The xfst Interface is a command line

- 1 open and process files
- 2 enter commands/REGULAR EXPRESSIONS directly

The STACK is shown within the square brackets

# A Short Look at the Xfst Interface

```
xfst[3]:
```

The xfst Interface is a command line

- 1 open and process files
- 2 enter commands/REGULAR EXPRESSIONS directly

The STACK is shown within the square brackets

→ it is a *last in, first out* data structure (LIFO)

# A Short Look at the Xfst Interface

```
xfst[3]:
```

The xfst Interface is a command line

- 1 open and process files
- 2 enter commands/`REGULAR EXPRESSIONS` directly

The `STACK` is shown within the square brackets

- it is a *last in, first out* data structure (LIFO)
- serves to store the different networks

# A Short Look at the Xfst Interface

`xfst[3]:`

The `xfst` Interface is a command line

- 1 open and process files
- 2 enter commands/`REGULAR EXPRESSIONS` directly

The `STACK` is shown within the square brackets

- it is a *last in, first out* data structure (LIFO)
- serves to store the different networks
- `STACK` operations will be introduced later

# The Goal of the Book is to teach ...

- ... linguists how to use the tools and techniques.



# The Goal of the Book is to teach ...

- ... linguists how to use the tools and techniques.
- ... the formal properties of finite state networks.

# The Goal of the Book is to teach ...

- ... linguists how to use the tools and techniques.
- ... the formal properties of finite state networks.
- ... to build useful and efficient programs that process text in natural languages.

# Applications of the software

- 1 **Tokenization** divides a running input text into tokens

# Applications of the software

- 1 **Tokenization** divides a running input text into tokens
- 2 Several finite state **morphological transducers** have been developed

# Applications of the software

- 1 **Tokenization** divides a running input text into tokens
- 2 Several finite state **morphological transducers** have been developed
- 3 **Disambiguation** of Tokens

# Applications of the software

- 1 **Tokenization** divides a running input text into tokens
- 2 Several finite state **morphological transducers** have been developed
- 3 **Disambiguation** of Tokens
- 4 Finite state **syntactic shallow parsers**

# Content

- Theory of Finite State Morphology
  - some facts — the book & the software
  - **some basic knowledge — finite state morphology**
  - building networks with xfst
    - The Lexicon
    - Regular Expressions
    - The Interface
  - possible applications
- Practical application of Finite State Morphology

# Morphological Application

We will concentrate on morphological application and the two central problems:



# Morphological Application

We will concentrate on morphological application and the two central problems:

- 1 WORD FORMATION (= Morphotactics/Morphosyntax): Words are composed of smaller units of meaning called *Morphemes*.

# Morphological Application

We will concentrate on morphological application and the two central problems:

- 1 WORD FORMATION (= Morphotactics/Morphosyntax): Words are composed of smaller units of meaning called *Morphemes*.
  - constrained to appear in certain combinations

# Morphological Application

We will concentrate on morphological application and the two central problems:

- 1 WORD FORMATION (= Morphotactics/Morphosyntax): Words are composed of smaller units of meaning called *Morphemes*.
  - constrained to appear in certain combinations
  - *piti-less-ness* vs. \**piti-ness-less*

# Morphological Application

We will concentrate on morphological application and the two central problems:

- 1 WORD FORMATION (= Morphotactics/Morphosyntax): Words are composed of smaller units of meaning called *Morphemes*.
  - constrained to appear in certain combinations
  - *piti-less-ness* vs. \**piti-ness-less*
- 2 PHONOLOGICAL/ORTHOGRAPHICAL ALTERNATIONS: spelling/sound of a morpheme often depends on its environment.

# Morphological Application

We will concentrate on morphological application and the two central problems:

- 1 WORD FORMATION (= Morphotactics/Morphosyntax): Words are composed of smaller units of meaning called *Morphemes*.
  - constrained to appear in certain combinations
  - *piti-less-ness* vs. \**piti-ness-less*
- 2 PHONOLOGICAL/ORTHOGRAPHICAL ALTERNATIONS: spelling/sound of a morpheme often depends on its environment.
  - *pity* is realized as *piti* in the context of a following *less*

# Morphological Application

We will concentrate on morphological application and the two central problems:

- ① WORD FORMATION (= Morphotactics/Morphosyntax): Words are composed of smaller units of meaning called *Morphemes*.
  - constrained to appear in certain combinations
  - *piti-less-ness* vs. \**piti-ness-less*
  
- ② PHONOLOGICAL/ORTHOGRAPHICAL ALTERNATIONS: spelling/sound of a morpheme often depends on its environment.
  - *pity* is realized as *piti* in the context of a following *less*
  - Therefore it is *piti-less* instead of \**pity-less*

# Finite State Networks — Acceptor

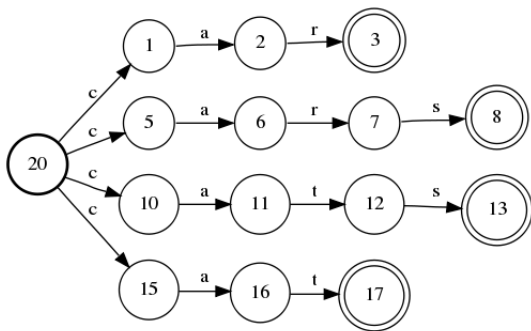
# Finite State Networks — Acceptor

Consider the following network for the words *car*, *cat*, *cars* and *cats*:



# Finite State Networks — Acceptor

Consider the following network for the words *car*, *cat*, *cars* and *cats*:



Accept.dot

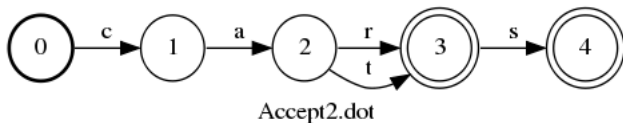
# Finite State Networks — optimized Acceptor

# Finite State Networks — optimized Acceptor

However, the paths and states can be shared...

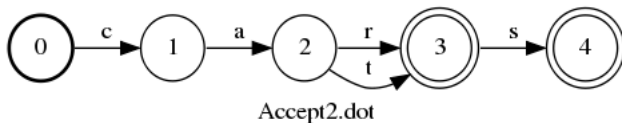
# Finite State Networks — optimized Acceptor

However, the paths and states can be shared...



# Finite State Networks — optimized Acceptor

However, the paths and states can be shared...



... and space and running time can be saved, which makes the networks rather quick.

# Finite State Networks — Acceptor



This type network is called an *Acceptor*.

# Finite State Networks — Acceptor



This type network is called an *Acceptor*.

- There is always an **upper** (analysis) and a **lower** (generation) side to the network

# Finite State Networks — Acceptor



This type network is called an *Acceptor*.

- There is always an **upper** (analysis) and a **lower** (generation) side to the network
- Within *Acceptors* the strings on both sides of the paths are the same (*Identity relation a:a*)



# Finite State Networks — Acceptor



This type network is called an *Acceptor*.

- There is always an **upper** (analysis) and a **lower** (generation) side to the network
- Within *Acceptors* the strings on both sides of the paths are the same (*Identity relation a:a*)
- This can be useful for spellchecking and the like.

# Finite State Networks — Acceptor



This type network is called an *Acceptor*.

- There is always an **upper** (analysis) and a **lower** (generation) side to the network
- Within *Acceptors* the strings on both sides of the paths are the same (*Identity relation a:a*)
- This can be useful for spellchecking and the like.
- In *xfst*-terms, this means:

# Finite State Networks — Acceptor



This type network is called an *Acceptor*.

- There is always an **upper** (analysis) and a **lower** (generation) side to the network
- Within *Acceptors* the strings on both sides of the paths are the same (*Identity relation a:a*)
- This can be useful for spellchecking and the like.
- In *xfst*-terms, this means:

```

xfst[1]:  up car
          car
  
```

# Finite State Networks — Acceptor



This type network is called an *Acceptor*.

- There is always an **upper** (analysis) and a **lower** (generation) side to the network
- Within *Acceptors* the strings on both sides of the paths are the same (*Identity relation a:a*)
- This can be useful for spellchecking and the like.
- In *xfst*-terms, this means:

```
xfst[1]: up car
         car
```

```
xfst[1]: down car
         car
```

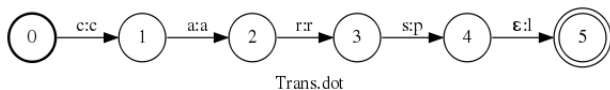
# Finite State Networks — Transducer

# Finite State Networks — Transducer

In order to work with morphological analysis/generation, *Transducers* are very useful:

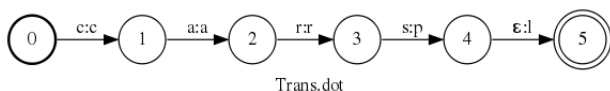
# Finite State Networks — Transducer

In order to work with morphological analysis/generation, *Transducers* are very useful:



# Finite State Networks — Transducer

In order to work with morphological analysis/generation, *Transducers* are very useful:

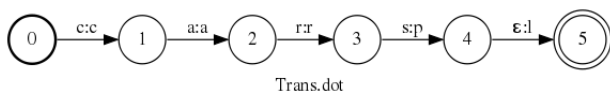


Transducers are two-sided, which makes morphological analysis possible.



# Finite State Networks — Transducer

In order to work with morphological analysis/generation, *Transducers* are very useful:

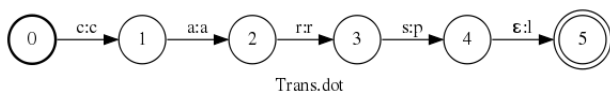


Transducers are two-sided, which makes morphological analysis possible.

In *xfst*-terms, this means:

# Finite State Networks — Transducer

In order to work with morphological analysis/generation, *Transducers* are very useful:



Transducers are two-sided, which makes morphological analysis possible.

In *xfst*-terms, this means:

```
xfst[1]:  up cars
          car+pl
```

```
xfst[1]:  down car+pl
          cars
```

# Content

- Theory of Finite State Morphology
  - some facts — the book & the software
  - some basic knowledge — finite state morphology
  - **building networks with xfst**
    - The Lexicon
    - Regular Expressions
    - The Interface
  - possible applications
- Practical application of Finite State Morphology

# Basic Ingredients

# Basic Ingredients

For an already very powerful finite state automata, one needs

# Basic Ingredients

For an already very powerful finite state automata, one needs

- 1 LEXICON:

# Basic Ingredients

For an already very powerful finite state automata, one needs

① LEXICON:

→ which contains the stems,

# Basic Ingredients

For an already very powerful finite state automata, one needs

① LEXICON:

- which contains the stems,
- the inflectional or derivational morphemes



# Basic Ingredients

For an already very powerful finite state automata, one needs

① LEXICON:

- which contains the stems,
- the inflectional or derivational morphemes
- and the appropriate morphological analysis

# Basic Ingredients

For an already very powerful finite state automata, one needs

① LEXICON:

- which contains the stems,
- the inflectional or derivational morphemes
- and the appropriate morphological analysis

② REGULAR EXPRESSIONS:

# Basic Ingredients

For an already very powerful finite state automata, one needs

① LEXICON:

- which contains the stems,
- the inflectional or derivational morphemes
- and the appropriate morphological analysis

② REGULAR EXPRESSIONS:

- which manipulate the forms of the lexicon on the basis of phonological rules

# Basic Ingredients

For an already very powerful finite state automata, one needs

① LEXICON:

- which contains the stems,
- the inflectional or derivational morphemes
- and the appropriate morphological analysis

② REGULAR EXPRESSIONS:

- which manipulate the forms of the lexicon on the basis of phonological rules

③ EXECUTABLE SCRIPT — short: SCRIPT:

# Basic Ingredients

For an already very powerful finite state automata, one needs

① LEXICON:

- which contains the stems,
- the inflectional or derivational morphemes
- and the appropriate morphological analysis

② REGULAR EXPRESSIONS:

- which manipulate the forms of the lexicon on the basis of phonological rules

③ EXECUTABLE SCRIPT — short: SCRIPT:

- which saves you a lot of typing

# The Lexicon

Basic idea: The LEXICON contains different *states* for each morpheme.

# The Lexicon

Basic idea: The LEXICON contains different *states* for each morpheme.

- It starts with the declaration of the morphological symbols, the *tags*:

# The Lexicon

Basic idea: The LEXICON contains different *states* for each morpheme.

- It starts with the declaration of the morphological symbols, the *tags*:

```
Multichar_Symbols
```

```
+Pl +Sg
```



# The Lexicon

Basic idea: The LEXICON contains different *states* for each morpheme.

- It starts with the declaration of the morphological symbols, the *tags*:

Multichar\_Symbols

+Pl +Sg

- followed by the different Lexicons, starting out with LEXICON ROOT, the *start state*:

# The Lexicon

Basic idea: The LEXICON contains different *states* for each morpheme.

- It starts with the declaration of the morphological symbols, the *tags*:

```
Multichar_Symbols
    +Pl +Sg
```

- followed by the different Lexicons, starting out with LEXICON ROOT, the *start state*:

```
LEXICON Root
    cat    SgPl;
    car    SgPl;
```

# The Lexicon

Basic idea: The LEXICON contains different *states* for each morpheme.

- It starts with the declaration of the morphological symbols, the *tags*:

```
Multichar_Symbols
    +Pl +Sg
```

- followed by the different Lexicons, starting out with LEXICON ROOT, the *start state*:

```
LEXICON Root
    cat    SgPl;
    car    SgPl;
```

Here, the stems are included.

The next Lexicon is indicated by the made-up *SGPL* at the end followed by a semicolon.

# The Lexicon

The Automata now jumps to the next state/lexicon called SGPL.

# The Lexicon

The Automata now jumps to the next state/lexicon called SGPL.

```
LEXICON SgPl
```

```
+Pl:s      #;
```

```
+Sg:0      #;
```

# The Lexicon

The Automata now jumps to the next state/lexicon called SGPL.

```
LEXICON SgPl
    +Pl:s      #;
    +Sg:0      #;
```

- The left side of the colon represents the *upper* (the analysis) side of the transducer.

# The Lexicon

The Automata now jumps to the next state/lexicon called SGPL.

```
LEXICON SgPl
    +Pl:s      #;
    +Sg:0      #;
```

- The left side of the colon represents the *upper* (the analysis) side of the transducer.
- The right side shows the *lower* side (the generation/surface form).

# The Lexicon

The Automata now jumps to the next state/lexicon called SGPL.

```
LEXICON SgPl
    +Pl:s      #;
    +Sg:0      #;
```

- The left side of the colon represents the *upper* (the analysis) side of the transducer.
- The right side shows the *lower* side (the generation/surface form).  
The surface morpheme on the right side is connected to the analysis on the left side.



# The Lexicon

The Automata now jumps to the next state/lexicon called SGPL.

```
LEXICON SgPl
    +Pl:s      #;
    +Sg:0      #;
```

- The left side of the colon represents the *upper* (the analysis) side of the transducer.
- The right side shows the *lower* side (the generation/surface form).

The surface morpheme on the right side is connected to the analysis on the left side.

In this case, +Pl is connected to [-s]. +Sg however, is represented by a Null-morpheme.

# The Lexicon

The Automata now jumps to the next state/lexicon called SGP1.

```
LEXICON SgP1
    +Pl:s      #;
    +Sg:0      #;
```

- The left side of the colon represents the *upper* (the analysis) side of the transducer.
- The right side shows the *lower* side (the generation/surface form).  
The surface morpheme on the right side is connected to the analysis on the left side.  
In this case, +Pl is connected to [-s]. +Sg however, is represented by a Null-morpheme.
- The hash symbol at the end of the row indicates the end of the path - this Lexicon is therefore the *final state*.

# The Lexicon - an Overview

# The Lexicon - an Overview

Multichar\_symbols

+Pl +Sg +N +A

# The Lexicon - an Overview

Multichar\_symbols

+Pl +Sg +N +A

LEXICON Root

milk            Noun;

car             Noun;

pity            Noun;

# The Lexicon - an Overview

Multichar\_symbols

+Pl +Sg +N +A

LEXICON Root

milk            Noun;

car             Noun;

pity            Noun;

LEXICON Noun

+N:0            SgPl;

Adj;

# The Lexicon - an Overview

Multichar\_symbols

+Pl +Sg +N +A

LEXICON Root

milk            Noun;

car             Noun;

pity            Noun;

LEXICON Noun

+N:0            SgPl;

Adj;

LEXICON SgPl

+Pl:s           #;

+Sg:0           #;

# The Lexicon - an Overview

Multichar\_symbols

+Pl +Sg +N +A

LEXICON Root

milk Noun;

car Noun;

pity Noun;

LEXICON Noun

+N:0 SgPl;

Adj;

LEXICON SgPl

+Pl:s #;

+Sg:0 #;

LEXICON Adj

+Adj:less #;



## The Script — Calling up the Lexicon

The `SCRIPT` (`script.xfst`) is a source code  
— avoids too much typing at the interface:

## The Script — Calling up the Lexicon

The `SCRIPT` (`script.xfst`) is a source code  
— avoids too much typing at the interface:

- Our first entry will be

```
clear
```

which ensures that there are no “leftovers” on the stack

## The Script — Calling up the Lexicon

The `SCRIPT` (`script.xfst`) is a source code  
— avoids too much typing at the interface:

- Our first entry will be

```
clear
```

which ensures that there are no “leftovers” on the stack

- In order to open up the `LEXICON`, we add:

```
read lexc < testlex.txt
```

## The Script — Calling up the Lexicon

The `SCRIPT` (`script.xfst`) is a source code  
— avoids too much typing at the interface:

- Our first entry will be

```
clear
```

which ensures that there are no “leftovers” on the stack

- In order to open up the `LEXICON`, we add:

```
read lexc < testlex.txt
```

Back to the `LEXICON`...

# The Lexicon — Overgeneration

However, the current LEXICON does allow a lot of overgeneration.

# The Lexicon — Overgeneration

However, the current LEXICON does allow a lot of overgeneration.

```
xfst[0]: read lexc < testlex.txt
Reading from 'testlex.txt'
Root...3, Noun...2, SgPl...2, Adj...1
Building lexicon...Minimizing...Done!
1.6 Kb. 15 states, 18 arcs, 9 paths.
Closing 'testlex.txt'
xfst[1]: print lower-words
milkless
milks
milk
carless
cars
car
pityless
pitys
pity
xfst[1]: □
```

# The Lexicon — Overgeneration

However, the current LEXICON does allow a lot of overgeneration.

```

xfst[0]: read lexc < testlex.txt
Reading from 'testlex.txt'
Root...3, Noun...2, SgPl...2, Adj...1
Building lexicon...Minimizing...Done!
1.6 Kb. 15 states, 18 arcs, 9 paths.
Closing 'testlex.txt'
xfst[1]: print lower-words
milkless
milks
milk
carless
cars
car
pityless
pitys
pity
xfst[1]: 

```

There are four words that should not exist:

# The Lexicon — Overgeneration

However, the current LEXICON does allow a lot of overgeneration.

```

xfst[0]: read lexc < testlex.txt
Reading from 'testlex.txt'
Root...3, Noun...2, SgPl...2, Adj...1
Building lexicon...Minimizing...Done!
1.6 Kb. 15 states, 18 arcs, 9 paths.
Closing 'testlex.txt'
xfst[1]: print lower-words
milkless
milks
milk
carless
cars
car
pityless
pitys
pity
xfst[1]: □

```

There are four words that should not exist:

- 1 **\*milkless** → The noun *milk* cannot become an adjective by means of the suffix *-less*



# The Lexicon — Overgeneration

However, the current LEXICON does allow a lot of overgeneration.

```

xfst[0]: read lexc < testlex.txt
Reading from 'testlex.txt'
Root...3, Noun...2, SgPl...2, Adj...1
Building lexicon...Minimizing...Done!
1.6 Kb. 15 states, 18 arcs, 9 paths.
Closing 'testlex.txt'
xfst[1]: print lower-words
milkless
milks
milk
carless
cars
car
pityless
pitys
pity
xfst[1]: □

```

There are four words that should not exist:

- ① **\*milkless** → The noun *milk* cannot become an adjective by means of the suffix *-less*
- ② **\*milks** and **\*pitys** → Both nouns are uncountable (no Plurals)

# The Lexicon — Overgeneration

However, the current LEXICON does allow a lot of overgeneration.

```

xfst[0]: read lexc < testlex.txt
Reading from 'testlex.txt'
Root...3, Noun...2, SgPl...2, Adj...1
Building lexicon...Minimizing...Done!
1.6 Kb. 15 states, 18 arcs, 9 paths.
Closing 'testlex.txt'
xfst[1]: print lower-words
milkless
milks
milk
carless
cars
car
pityless
pitys
pity
xfst[1]: □

```

There are four words that should not exist:

- ① **\*milkless** → The noun *milk* cannot become an adjective by means of the suffix *-less*
- ② **\*milks** and **\*pitys** → Both nouns are uncountable (no Plurals)
- ③ **\*pityless** → phonological rule is needed for correct spelling: *pitiless*.

# The Lexicon — Flags

1. and 2. can be solved by manipulating the LEXICON itself.

There are two possibilities:

# The Lexicon — Flags

1. and 2. can be solved by manipulating the LEXICON itself.

There are two possibilities:

- 1 More paths/lexicons can be added

# The Lexicon — Flags

1. and 2. can be solved by manipulating the LEXICON itself.

There are two possibilities:

- 1 More paths/lexicons can be added
- 2 So-called FLAG DIACRITICS can be integrated

# The Lexicon — Flags

1. and 2. can be solved by manipulating the LEXICON itself.

There are two possibilities:

- 1 More paths/lexicons can be added
- 2 So-called FLAG DIACRITICS can be integrated

These flags can be imagined as invisible markers that are added to strings. Other flags are stop signs, which will allow only certain strings and their flags to pass through.

# The Lexicon — Types of Flags

Generally, flags are split into three parts:

# The Lexicon — Types of Flags

Generally, flags are split into three parts:

@NAME.FEATURE.VALUE@



# The Lexicon — Types of Flags

Generally, flags are split into three parts:

`@NAME.FEATURE.VALUE@`

- **NAME:** Flag-names usually consist of only one letter, three of the possibilities are listed below

# The Lexicon — Types of Flags

Generally, flags are split into three parts:

`@NAME.FEATURE.VALUE@`

- **NAME:** Flag-names usually consist of only one letter, three of the possibilities are listed below
  - P → Positive: marks the flag as carrying that specific feature-value pair

# The Lexicon — Types of Flags

Generally, flags are split into three parts:

`@NAME.FEATURE.VALUE@`

- **NAME:** Flag-names usually consist of only one letter, three of the possibilities are listed below
  - P → Positive: marks the flag as carrying that specific feature-value pair
  - R → Requires that feature-value pair on a string to open up this path

# The Lexicon — Types of Flags

Generally, flags are split into three parts:

`@NAME.FEATURE.VALUE@`

- **NAME:** Flag-names usually consist of only one letter, three of the possibilities are listed below
  - P → Positive: marks the flag as carrying that specific feature-value pair
  - R → Requires that feature-value pair on a string to open up this path
  - D → Disallows that specific feature-value pair on a string to pass

# The Lexicon — Types of Flags

Generally, flags are split into three parts:

@NAME.FEATURE.VALUE@

- **NAME:** Flag-names usually consist of only one letter, three of the possibilities are listed below
  - P → Positive: marks the flag as carrying that specific feature-value pair
  - R → Requires that feature-value pair on a string to open up this path
  - D → Disallows that specific feature-value pair on a string to pass
- **FEATURE:** The features can be invented individually. They often describe a certain grammatical category like *case* or *number*.

# The Lexicon — Types of Flags

Generally, flags are split into three parts:

@NAME.FEATURE.VALUE@

- **NAME:** Flag-names usually consist of only one letter, three of the possibilities are listed below
  - P → Positive: marks the flag as carrying that specific feature-value pair
  - R → Requires that feature-value pair on a string to open up this path
  - D → Disallows that specific feature-value pair on a string to pass
- **FEATURE:** The features can be invented individually. They often describe a certain grammatical category like *case* or *number*.
- **VALUE:** Values are the specific shapes of the grammatical categories, e.g. *Sg*, *Pl*, *Dat* or *Acc*.

# The Lexicon — Flags

The Lexicon with FLAG DIACRITICS:

# The Lexicon — Flags

## The Lexicon with FLAG DIACRITICS:

```

Multichar_Symbols

+Sg +Pl +N +A
@P.LIQUID.yes@ @D.LIQUID@
@P.COUNT.yes@ @R.COUNT.yes@

LEXICON Root

milk@P.LIQUID.yes@      Noun;
car@P.COUNT.yes@       Noun;
pity                    Noun;

LEXICON Noun
+N:0                    SqPl;
@D.LIQUID@             Adj;

LEXICON SgPl

< "+Pl":s "@R.COUNT.yes@" >  #;
+Sg:0                   #;

LEXICON Adj
+A:less                 #;

```



# The Lexicon — Flags

## The Lexicon with FLAG DIACRITICS:

```

Multichar_Symbols

+Sg +Pl +N +A
@P.LIQUID.yes@ @D.LIQUID@
@P.COUNT.yes@ @R.COUNT.yes@

LEXICON Root

milk@P.LIQUID.yes@      Noun;
car@P.COUNT.yes@      Noun;
pity                    Noun;

LEXICON Noun
+N:0                    SqPl;
@D.LIQUID@             Adj;

LEXICON SgPl

< "+Pl":s "@R.COUNT.yes@" > #;
+Sg:0                  #;

LEXICON Adj
+A:less                 #;

```

**However**, whenever you include flags into your LEXICON, remember to eliminate these after your compilation...

# The Lexicon — Flag Elimination

.... but why?

# The Lexicon — Flag Elimination

.... but why?

→ difficult manipulation via REGULAR EXPRESSIONS

# The Lexicon — Flag Elimination

.... but why?

- difficult manipulation via REGULAR EXPRESSIONS
- consider the following output:

# The Lexicon — Flag Elimination

... but why?

- difficult manipulation via REGULAR EXPRESSIONS
- consider the following output:

```

xfst[0]: read lexc < testlex.txt
Reading from 'testlex.txt'
Root...3, Noun...2, SgPl...2, Adj...1
Building lexicon...Minimizing...Done!
1.7 Kb. 19 states, 22 arcs, 9 paths.
Closing 'testlex.txt'
xfst[1]: print lower-words
milk
cars
car
carless
pity
pityless
xfst[1]: set show-flags ON
variable show-flags = ON
xfst[1]: print lower-words
milk@P.LIQUID.yes@
car@P.COUNT.yes@S@R.COUNT.yes@
car@P.COUNT.yes@
car@P.COUNT.yes@D.LIQUID@less
pity
pity@D.LIQUID@less
xfst[1]: eliminate flag COUNT
1.7 Kb. 19 states, 23 arcs, 7 paths.
xfst[1]: eliminate flag LIQUID
1.6 Kb. 18 states, 22 arcs, 6 paths.
xfst[1]: print lower-words
pityless
pity
milk
cars
car
carless
xfst[1]: []

```

# The Script — Eliminating flags

We already have two commands in our `script.xfst`:

```
clear  
read lexc < testlex.txt
```

# The Script — Eliminating flags

We already have two commands in our `script.xfst`:

```
clear  
read lexc < testlex.txt
```

- To eliminate the flags simply list the different features:

# The Script — Eliminating flags

We already have two commands in our `script.xfst`:

```
clear  
read lexc < testlex.txt
```

- To eliminate the flags simply list the different features:

```
eliminate flag LIQUID
```



## The Script — Eliminating flags

We already have two commands in our `script.xfst`:

```
clear  
read lexc < testlex.txt
```

- To eliminate the flags simply list the different features:

```
eliminate flag LIQUID  
eliminate flag COUNT
```

# The Script — Eliminating flags

We already have two commands in our `script.xfst`:

```
clear  
read lexc < testlex.txt
```

- To eliminate the flags simply list the different features:

```
eliminate flag LIQUID  
eliminate flag COUNT
```

- This will cover all of our four flags

## The Lexicon — Output with Flags

With flags, the output of our current LEXICON will look like the following:

# The Lexicon — Output with Flags

With flags, the output of our current LEXICON will look like the following:

```

xfst[1]: source script.xfst
Opening file script.xfst...
Reading from 'testlex.txt'
Root...3, Noun...2, SgPl...2, Adj...1
Building lexicon... Minimizing... Done!
1.7 Kb. 19 states, 22 arcs, 9 paths.
Closing 'testlex.txt'
1.7 Kb. 18 states, 21 arcs, 8 paths.
1.6 Kb. 18 states, 22 arcs, 6 paths.
Closing file script.xfst...
xfst[1]: print lower-words
pity
pityless
milk
cars
car
carless
xfst[1]: print upper-words
pity+N+Sg
pity+A
milk+N+Sg
car+N+Pl
car+N+Sg
car+A
xfst[1]: []

```

# The Lexicon — Output with Flags

With flags, the output of our current LEXICON will look like the following:

```

xfst[1]: source script.xfst
Opening file script.xfst...
Reading from 'testlex.txt'
Root...3, Noun...2, SgPl...2, Adj...1
Building lexicon...Minimizing...Done!
1.7 Kb. 19 states, 22 arcs, 9 paths.
Closing 'testlex.txt'
1.7 Kb. 18 states, 21 arcs, 8 paths.
1.6 Kb. 18 states, 22 arcs, 6 paths.
Closing file script.xfst...
xfst[1]: print lower-words
pity
pityless
milk
cars
car
carless
xfst[1]: print upper-words
pity+N+Sg
pity+A
milk+N+Sg
car+N+Pl
car+N+Sg
car+A
xfst[1]: []

```

**\*milks**, **\*pitys** and **\*milkless** have disappeared...

# The Lexicon — Output with Flags

With flags, the output of our current LEXICON will look like the following:

```

xfst[1]: source script.xfst
Opening file script.xfst...
Reading from 'testlex.txt'
Root...3, Noun...2, SgPl...2, Adj...1
Building lexicon... Minimizing... Done!
1.7 Kb. 19 states, 22 arcs, 9 paths.
Closing 'testlex.txt'
1.7 Kb. 18 states, 21 arcs, 8 paths.
1.6 Kb. 18 states, 22 arcs, 6 paths.
Closing file script.xfst...
xfst[1]: print lower-words
pity
pityless
milk
cars
car
carless
xfst[1]: print upper-words
pity+N+Sg
pity+A
milk+N+Sg
car+N+Pl
car+N+Sg
car+A
xfst[1]: []

```

**\*milks**, **\*pitys** and **\*milkless** have disappeared...

... but what about *\*pityless* ???

# Content

- Theory of Finite State Morphology
  - some facts — the book & the software
  - some basic knowledge — finite state morphology
  - building networks with xfst
    - The Lexicon
    - **Regular Expressions**
    - The Interface
  - possible applications
- Practical application of Finite State Morphology

# Regular Expressions

*\*pityless* needs to be dealt with by REGULAR EXPRESSIONS.



# Regular Expressions

*\*pityless* needs to be dealt with by REGULAR EXPRESSIONS.

Languages that can be described in finite state are those, which can be described by REGULAR EXPRESSIONS.

# Regular Expressions

*\*pityless* needs to be dealt with by REGULAR EXPRESSIONS.

Languages that can be described in finite state are those, which can be described by REGULAR EXPRESSIONS.

- describes a string **a** (for a simple acceptor) or

# Regular Expressions

*\*pityless* needs to be dealt with by REGULAR EXPRESSIONS.

Languages that can be described in finite state are those, which can be described by REGULAR EXPRESSIONS.

- describes a string **a** (for a simple acceptor) or
- a relation **a:a** (for a transducer) and

# Regular Expressions

*\*pityless* needs to be dealt with by REGULAR EXPRESSIONS.

Languages that can be described in finite state are those, which can be described by REGULAR EXPRESSIONS.

- describes a string **a** (for a simple acceptor) or
- a relation **a:a** (for a transducer) and
- can be compiled into a finite network

# Regular Expressions

Some basic REGULAR EXPRESSIONS — mainly following classical computer science:

$\emptyset$ = EPSILON	$?$ = ANY SYMBOL	$\#$ = BOUNDARY SYMBOL
$()$ = OPTIONALITY	$+$ = CONCATENATION WITH ITSELF ONE OR MORE TIMES	$*$ = CONCATENATION WITH ITSELF ZERO OR MORE TIMES
$\sim$ = NEGATION	$_$ = PLACE HOLDER	$\{ \}$ AND $\cdot$ = CONCATENATION
$[]$ = GROUPING	$\rightarrow$ = BECOMES...	$  $ = IN THE CONTEXT OF...
$ $ = UNION	$\&$ = INTERSECTION	$\cdot$ AND $:$ = CROSSPRODUCT
$\circ$ = COMPOSITION		

# Regular Expressions — An Example

With an abstract example like:

$$[ \{ab\} c .x. \{de\} f* g ]$$

what would I get as output?

# Regular Expressions — An Example

With an abstract example like:

$$[ \{ab\} c .x. \{de\} f^* g ]$$

what would I get as output?

```
xfst[0]: read regex [ {ab} c .x. {de} f* g];
264 bytes. 5 states, 6 arcs, Circular.
xfst[1]: print words
<a:d><b:e><c:g>
<a:d><b:e><c:f><0:g>
xfst[1]: print random-lower
defffg
deg
deffg
defffg
defffg
deg
deg
deg
deg
deg
deffg
deffg
deffffg
.
```

```
xfst[i]: print random-upper
abc
abc
abc
abc
abc
abc
abc
abc
abc
abc
abc
abc
abc
abc
abc
abc
abc
abc
abc
abc
xfst[1]: []
```

## Regular Expressions — *\*pityless*

REGULAR EXPRESSIONS are mostly used to manipulate the LEXICON phonologically.

In the case of *\*pityless*, the phonological rule would be:

$$[ y \rightarrow i \mid \_ l e s s ]$$

which translates as

*'y' becomes 'i' iff 'less' follows 'y'*



## Regular Expressions — The File

The phonological rules (and other `REGULAR EXPRESSIONS`) are kept in a separate file: `testrules.regex`. Here, they are used by means of composition:

## Regular Expressions — The File

The phonological rules (and other `REGULAR EXPRESSIONS`) are kept in a separate file: `testrules.regex`. Here, they are used by means of composition:

Regular Expression 1

.o.

Regular Expression 2

.o.

Regular Expression 3;

## Regular Expressions — The File

The phonological rules (and other REGULAR EXPRESSIONS) are kept in a separate file: `testrules.regex`. Here, they are used by means of composition:

Regular Expression 1

.o.

Regular Expression 2

.o.

Regular Expression 3;

- Be aware of the fact that the second rule will take as the input the output of the first rule etc. (*feeding and bleeding*).

## Regular Expressions — The File

The phonological rules (and other REGULAR EXPRESSIONS) are kept in a separate file: `testrules.regex`. Here, they are used by means of composition:

Regular Expression 1

.o.

Regular Expression 2

.o.

Regular Expression 3;

- Be aware of the fact that the second rule will take as the input the output of the first rule etc. (*feeding and bleeding*).
- The correct succession of phonological rules is therefore of great importance.

# The Script — Introducing the Regular Expressions

We already have four commands in our script.xfst:

```
clear  
read lexc < testlex.txt  
eliminate flag LIQUID  
eliminate flag COUNT
```

# The Script — Introducing the Regular Expressions

We already have four commands in our `script.xfst`:

```
clear
read lexc < testlex.txt
eliminate flag LIQUID
eliminate flag COUNT
```

In order to introduce the `REGULAR EXPRESSIONS` we need to add another entry:

# The Script — Introducing the Regular Expressions

We already have four commands in our `script.xfst`:

```
clear
read lexc < testlex.txt
eliminate flag LIQUID
eliminate flag COUNT
```

In order to introduce the `REGULAR EXPRESSIONS` we need to add another entry:

```
read regex < testrules.regex
```

# Content

- Theory of Finite State Morphology
  - some facts — the book & the software
  - some basic knowledge — finite state morphology
  - building networks with xfst
    - The Lexicon
    - Regular Expressions
    - **The Interface**
  - possible applications
- Practical application of Finite State Morphology



# The Interface

The `xfst-INTERFACE` is used to interact with the different files but also to accomplish smaller tasks directly.

# The Interface

The `xfst-INTERFACE` is used to interact with the different files but also to accomplish smaller tasks directly.

- The *stack* takes the networks as they come: the last one is on top.

```
xfst[3]:
```

# The Interface

The `xfst-INTERFACE` is used to interact with the different files but also to accomplish smaller tasks directly.

- The *stack* takes the networks as they come: the last one is on top.

`xfst[3]:`

- There are certain *stack operations* that help to manipulate the network:

# The Interface

The `xfst-INTERFACE` is used to interact with the different files but also to accomplish smaller tasks directly.

- The *stack* takes the networks as they come: the last one is on top.

`xfst[3]:`

- There are certain *stack operations* that help to manipulate the network:
  - 1 `pop stack` → will take away the top network

# The Interface

The `xfst-INTERFACE` is used to interact with the different files but also to accomplish smaller tasks directly.

- The *stack* takes the networks as they come: the last one is on top.

`xfst[3]:`

- There are certain *stack operations* that help to manipulate the network:
  - 1 `pop stack` → will take away the top network
  - 2 `turn stack` → will turn the stack around

# The Interface

The `xfst-INTERFACE` is used to interact with the different files but also to accomplish smaller tasks directly.

- The *stack* takes the networks as they come: the last one is on top.

`xfst[3]:`

- There are certain *stack operations* that help to manipulate the network:
  - 1 `pop stack` → will take away the top network
  - 2 `turn stack` → will turn the stack around
  - 3 `apply up/down word` → analyse/generate a certain string

# The Interface

The `xfst-INTERFACE` is used to interact with the different files but also to accomplish smaller tasks directly.

- The *stack* takes the networks as they come: the last one is on top.

`xfst[3]:`

- There are certain *stack operations* that help to manipulate the network:
  - 1 `pop stack` → will take away the top network
  - 2 `turn stack` → will turn the stack around
  - 3 `apply up/down word` → analyse/generate a certain string
  - 4 `compose/concatenate/union net` → see REGULAR EXPRESSIONS

# The Interface

The `xfst-INTERFACE` is used to interact with the different files but also to accomplish smaller tasks directly.

- The *stack* takes the networks as they come: the last one is on top.

`xfst[3]:`

- There are certain *stack operations* that help to manipulate the network:
  - 1 pop stack → will take away the top network
  - 2 turn stack → will turn the stack around
  - 3 apply up/down *word* → analyse/generate a certain string
  - 4 compose/concatenate/union net → see REGULAR EXPRESSIONS
- In order to combine our rules and our lexicon, we need the *composition-operator*



# The Interface — Composing our Networks

Some thoughts about **Composition**

# The Interface — Composing our Networks

## Some thoughts about **Composition**

- Our `testrules.regex` need an input they can work with

# The Interface — Composing our Networks

## Some thoughts about **Composition**

- Our `testrules.regex` need an input they can work with
- Therefore they need to be at the second position of the composing process

# The Interface — Composing our Networks

## Some thoughts about **Composition**

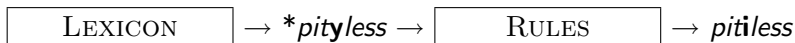
- Our `testrules.regex` need an input they can work with
- Therefore they need to be at the second position of the composing process

`LEXICON` → `*pityless` → `RULES` → `pityless`

# The Interface — Composing our Networks

## Some thoughts about **Composition**

- Our `testrules.regex` need an input they can work with
- Therefore they need to be at the second position of the composing process

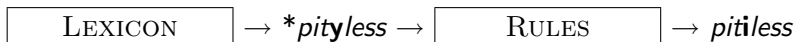


- For the `LEXICON` to be dealt with first by the *composition operator*, it must be on top of the stack

# The Interface — Composing our Networks

## Some thoughts about **Composition**

- Our `testrules.regex` need an input they can work with
- Therefore they need to be at the second position of the composing process



- For the `LEXICON` to be dealt with first by the *composition operator*, it must be on top of the stack
- We therefore need to adjust our `SCRIPT`

# The Script — Last Adjustments

First, we add

# The Script — Last Adjustments

First, we add

- `turn stack` to our SCRIPT.



# The Script — Last Adjustments

First, we add

- turn `stack` to our `SCRIPT`.

We compose the two networks and add

# The Script — Last Adjustments

First, we add

- `turn stack` to our `SCRIPT`.

We compose the two networks and add

- `compose net` which gives us a final `SCRIPT`:

## The Script — Last Adjustments

First, we add

- turn stack to our SCRIPT.

We compose the two networks and add

- compose net which gives us a final SCRIPT:

```
clear
read lexc < testlex.txt
eliminate flag LIQUID
eliminate flag COUNT
read regex < testrules.regex
turn stack
compose net
```

# The Interface — Final Output

```

xfst[0]: source script.xfst
Opening file script.xfst...
Reading from 'testlex.txt'
Root...3, Noun...2, SgPl...2, Adj...1
Building lexicon...Minimizing...Done!
1.7 Kb. 19 states, 22 arcs, 9 paths.
Closing 'testlex.txt'
1.7 Kb. 18 states, 21 arcs, 8 paths.
1.6 Kb. 18 states, 22 arcs, 6 paths.
Opening file testrules.regex...
1.7 Kb. 9 states, 38 arcs, Circular.
Closing file testrules.regex...
1.7 Kb. 18 states, 22 arcs, 6 paths.
Closing file script.xfst...
xfst[1]: print lower-words
pity
pitiless
milk
cars
car
carless
xfst[1]: up pitiless
pity+A
xfst[1]: □

```

# Content

- Theory of Finite State Morphology
  - some facts — the book & the software
  - some basic knowledge — finite state morphology
  - building networks with xfst
    - The Lexicon
    - Regular Expressions
    - The Interface
  - **possible applications**
- Practical application of Finite State Morphology

# Possibilities of Finite-State-Morphology

Just to give you an overview on how far the power of Finite State Morphology reaches:

# Possibilities of Finite-State-Morphology

Just to give you an overview on how far the power of Finite State Morphology reaches:

- Prefixes, suffixes and stem alternations

# Possibilities of Finite-State-Morphology

Just to give you an overview on how far the power of Finite State Morphology reaches:

- Prefixes, suffixes and stem alternations
- Restricted reduplication (e.g. Tagalog: *kukuha* — “take”)



# Possibilities of Finite-State-Morphology

Just to give you an overview on how far the power of Finite State Morphology reaches:

- Prefixes, suffixes and stem alternations
- Restricted reduplication (e.g. Tagalog: *kukuha* — “take”)
- Full stem reduplication (e.g. in Malay: *buku* — “book”;  
*buku-buku* — “books”)

# Possibilities of Finite-State-Morphology

Just to give you an overview on how far the power of Finite State Morphology reaches:

- Prefixes, suffixes and stem alternations
- Restricted reduplication (e.g. Tagalog: *kukuha* — “take”)
- Full stem reduplication (e.g. in Malay: *buku* — “book”;  
*buku-buku* — “books”)
- Semitic stem interdigitation (e.g. Arabic )

Thank You for Listening!

# Thank You for Listening!

Now turning to the practical part of the tutorial...

Login name: fast  
Password: fast  
Log on to: (this computer)

Please do not forget to constantly save  
EVERYTHING you do  
in case the light goes...