## Regular Expressions

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[A, B]</td>
<td>A followed by B</td>
</tr>
<tr>
<td>{A, B}</td>
<td>A or B</td>
</tr>
<tr>
<td>[A, B^]</td>
<td>An A optionally followed by a B</td>
</tr>
<tr>
<td>A*</td>
<td>zero or more occurrences of A</td>
</tr>
<tr>
<td>A+</td>
<td>one or more occurrences of A</td>
</tr>
<tr>
<td>?</td>
<td>Any symbol</td>
</tr>
<tr>
<td>'0'..'9'</td>
<td>Symbol in the range of '0' .. '9'</td>
</tr>
<tr>
<td>$ A</td>
<td>A string containing A</td>
</tr>
</tbody>
</table>

## More Regular Expressions

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>~ A</td>
<td>A string not matching A</td>
</tr>
<tr>
<td>A - B</td>
<td>A string matching A but not B</td>
</tr>
<tr>
<td>A &amp; B</td>
<td>A string matching A and B</td>
</tr>
</tbody>
</table>
Examples

<table>
<thead>
<tr>
<th>Regex</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>[? *, i,s,h]</code></td>
<td>A string with suffix <em>ish</em></td>
</tr>
<tr>
<td><code>$ [q,u]</code></td>
<td>A string containing <em>qu</em></td>
</tr>
<tr>
<td><code>’0’..’9’</code></td>
<td>a digit</td>
</tr>
<tr>
<td><code>’0’..’9’ - ’2’</code></td>
<td>all digits except 2</td>
</tr>
<tr>
<td><code>~ ’0’ . . . ’9’</code></td>
<td>not a digit (i.e. includes a, 10, ε)</td>
</tr>
<tr>
<td><code>$ a &amp; $ b</code></td>
<td>strings containing an a and a b</td>
</tr>
</tbody>
</table>

From Reg Ex to FSA

- Every Reg Ex corresponds with a FS automaton
- Every Reg Ex operator defines an operation on FS automata

Epsilon

- Epsilon transition’s (jumps) allow transition from one state to another without reading any input symbol

Optional

- `[a,b]`
Kleene Closure

\[[a, b] \ast\]

Concatenation

\[[[a+, b+], [c+, d+]]\]

Union

\{[a+, b+], [c+, b+]\}

Complement

\[\epsilon\]

• Input automaton must be deterministic
Deterministic Recognizer

- A FS recognizer is deterministic iff
  - it has a single start state,
  - it has no epsilon transitions,
  - for each state and each symbol there is at most one applicable transition.

- For every $M$ there is a deterministic (efficient) automaton $M'$ such that $L(M) = L(M')$.

Removing Non-deterministic Transitions

Removing Epsilons

Converting NFA to DFA

www.cs.may.ie/~jpower/Courses/parsing/

We use a Deterministic Finite-State Automaton (DFA) which is a special case of a NFA with the additional requirements that:

- There are no transitions involving $\epsilon$,
- No state has two outgoing transitions based on the same symbol.
Subset Construction Algorithm

- The ε-closure function takes a state and returns the set of states reachable from it based on (one or more) ε-transitions.
- The function \( \text{move} \) takes a state and a character, and returns the set of states reachable by one transition on this character.

\[
\text{move}(\{A, B\}, a) = \text{move}(A, a) \cup \text{move}(B, a)
\]

The Subset Construction Algorithm II

1. DFA start state = ε-closure(NFA start state).
2. For each new DFA-state \( S \) and possible input symbol \( a \):
   - Add the transition \((S, a, \text{ε-closure}(\text{move}(S, a)))\)
3. Apply step 2 to newly added states.
4. DFA finish states = states containing a NFA finish state.

Example

\[
\{(0, a, 3), (3, b, 1), (3, b, 2), (1, b, 1), (2, c, 2)\}
\]

\[
\Downarrow
\{(0, a, 3), (3, b, \{1, 2\}), (\{1, 2\}, b, 1), (\{1, 2\}, c, 2), (1, b, 1), (2, c, 2)\}
\]

DFA
Intermezzo: 
RegEx without Kleene *

• Automata for languages definable without Kleene * or + have interesting properties (Yli Jyrää, EACL 2003)

• Can you define the language a* without using Kleene *, +, or $

Recognizers vs Transducers

• A finite state recognizer is an automaton which accepts strings (yes/no decisions):
  * recognize Zip Codes, Proper Names, Syllables, ...

• A finite state transducer is an automaton which maps one string onto another string:
  * Map Letters onto Phonemes, Inflected words onto Base Forms, Words onto Part of Speech Tags, ....

Stemming

• Translate a word into its base form,

• For information retrieval:
  * Given a query, find relevant documents
  * A query with republican, can lead to a document with republicans.
Stemming

Georgia → georgia
Republicans → republican
are → be
getting → get
strong → strong
encouragement → encouragement
to → to
enter → enter
a → a
candidate → candidate

Part of Speech Tagging

- Translate a sequence of words into a sequence of Part of Speech Tags
- Useful as a first step towards full parsing or to support searching for linguistic patterns,

Part of Speech Tagging

| AT1 a | JJ relative |
| NN1c handful | IO of |
| DAz such | NN2 reports |
| VBDZ was | VVNV received |

Grapheme to Phoneme Conversion

- Translate a sequence of letters into a sequence of phonemes
- Required for Text to Speech applications
- Each letter or sequence of letters is translated into a phoneme

```
a b b r e v i a t e d
↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓
@ b ε r i v l 1 t l d
```
Encoding a Rule

• $e \rightarrow I / \{t,d\} \rightarrow d \#$

RegEx Notation for Transducers

• $[a:b, c^*]$ translates, among others, $accc$ in $bccc$.

• $:$ is the ‘pair’-operator: it translates a symbol $A$ in a symbol $B$.

• abbreviated# $\rightarrow$ abbreviatId#

Regex Notation for Transducers

• $[a:b, c^*]$ is short for $[a:b, (c:c)^*]$

• By default, a regular expression without $:'$ is read as the identity-transducer: every symbol in the input is mapped onto itself.

Dutch Diminitives

<table>
<thead>
<tr>
<th></th>
<th>huis+je</th>
<th>haan+je</th>
<th>man+je</th>
</tr>
</thead>
<tbody>
<tr>
<td>input</td>
<td>huis+je</td>
<td>haan+je</td>
<td>man+je</td>
</tr>
<tr>
<td>output</td>
<td>huisje</td>
<td>haantje</td>
<td>mannetje</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>huis+je</th>
<th>haan+je</th>
<th>man+je</th>
</tr>
</thead>
<tbody>
<tr>
<td>input</td>
<td>huis+je</td>
<td>haan+je</td>
<td>man+je</td>
</tr>
<tr>
<td>output</td>
<td>huisje</td>
<td>haantje</td>
<td>mannetje</td>
</tr>
</tbody>
</table>
Dutch Dimunitives

[? *, {{s, + : []}, [a, a, n, + : t], [~a, a, n, [] : n, [] : e, + : t]}, j, e]}

(Non-)Determinism

- An transducer is deterministic if for every state and input symbol, at most a single transduction to a new state is possible.
- Non-deterministic transducers can sometimes be made deterministic, but not always.
- Non-deterministic recognizers can always be made deterministic.

Non-Determinism: Example

maan+je → maantje
man+je → mannetje
Two Sources of Non-determinism

- Unbounded Look-ahead
  - $\text{acccb} \rightarrow \text{bcccb} \quad \text{acccd} \rightarrow \text{dcccd}$
  - $\{[a:b, c^*, b], [a:d, c^*, d]\}$

- Multiple outputs
  - $\text{bloem+je} \rightarrow \text{bloempje}$
  - $\text{bloem+je} \rightarrow \text{bloemetje}$
  - $[?*, o, e, m, {+:p, +:[e,t]}], j, e$

Deterministic Transducers

- Deterministic transducers are more efficient than non-deterministic transducers (because no choice-points/backtracking/search is required).

- But deterministic transducers can be much larger than corresponding non-deterministic transducer.

- (t_determinize option in FSA).

Making a Transducer Deterministic

- $\text{acb} \rightarrow \text{bcb}$
- $\text{acd} \rightarrow \text{dcd}$

From English to Dutch Numbers

- Automatic translation of (spoken) English into Dutch requires translation of number words,

- twentyone $\rightarrow$ eenentwintig,

- twentyone $\rightarrow$ 21 $\rightarrow$ eenentwintig
From Number Words to Numbers

macro(one, {one:1, two:2, ..., nine:9 } ).
macro(twenty, {twenty:2, thirty:3, ..., ninety:9 } ).
macro(eng2num,{ one,ten:[1,0],
                  eleven:[1,1],..., 
                  nineteen:[1,9], 
                  [twenty,one] } ).

From English to Dutch Numbers

• Transducer T1 for translating English Number Words into Numbers,
• Transducer T2 for translating Numbers into Dutch Number Words
• The output of T1 is used as input by T2.

Composition

• The composition of transducers T1 and T2 is a new transducer T3, which is equivalent to passing the input through T1, taking the output of T1 as input for T2, and taking the output of T2 as output.
• T1 o T2 denotes the composition of T1 and T2.

Number Translation by Composition

macro(eng2num,   
        {{one,ten:[1,0],..}}).
macro(num2dut,   
        {1:een,2:twee, ....}).
macro(eng2dut,   
        eng2num o num2dut).
Input/Output reversal

• The inverse of a transducer $T$ is a transducer which takes as input the output of $T$, and produces as output the input of $T$.

• In FSA: $\text{inverse}(T)$.

• Translating Dutch into English:

macro(dutch2eng,
    inverse(num2dut) o inverse(eng2num) ).

Finite State POS Tagging

• Assign Part of Speech tags to words,

• but many words have more than one POS:

☆ The/det report/n was/aux written/v
☆ The/det police/n has/aux to/aux report/v
   all/det problems/n

Finite State POS Tagging

• A Solution:

☆ A non-deterministic $T$ which assigns a word all possible POS tags,
☆ Recognizers $R$ which filter the output of $T$,
☆ Compose $T$ and (the identity transducer for) $R$.

Finite State POS Tagging

macro(lexicon,
    { all:det,has:aux,police:n,problems:n,report:{v,n},the:det,to:v,was:aux,
        written:v}* ).
macro(no_det_v,
    $ [ \text{det}, v ]$ ).
macro(tagger,
    lexicon o no_det_v ).

Finite State POS Tagging

macro(dutch2eng,
    inverse(num2dut) o inverse(eng2num) ).