Language Processing with Perl and Prolog
Chapter 2: Corpus Processing Tools

Pierre Nugues

Lund University
Pierre.Nugues@cs.lth.se
http://cs.lth.se/pierre_nugues/
Corpora

A corpus is a collection of texts (written or spoken) or speech. Corpora are balanced from different sources: news, novels, etc.

<table>
<thead>
<tr>
<th>Most frequent words in a collection of contemporary running texts</th>
<th>English</th>
<th>French</th>
<th>German</th>
</tr>
</thead>
<tbody>
<tr>
<td>the</td>
<td>de</td>
<td>der</td>
<td></td>
</tr>
<tr>
<td>of</td>
<td>le (article)</td>
<td>die</td>
<td></td>
</tr>
<tr>
<td>to</td>
<td>la (article)</td>
<td>und</td>
<td></td>
</tr>
<tr>
<td>in</td>
<td>et</td>
<td>in</td>
<td></td>
</tr>
<tr>
<td>and</td>
<td>les</td>
<td>des</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Most frequent words in Genesis</th>
<th>English</th>
<th>French</th>
<th>German</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>et</td>
<td>und</td>
<td></td>
</tr>
<tr>
<td>the</td>
<td>de</td>
<td>die</td>
<td></td>
</tr>
<tr>
<td>of</td>
<td>la</td>
<td>der</td>
<td></td>
</tr>
<tr>
<td>his</td>
<td>à</td>
<td>da</td>
<td></td>
</tr>
<tr>
<td>he</td>
<td>il</td>
<td>er</td>
<td></td>
</tr>
</tbody>
</table>
Characteristics of Current Corpora

Big: The Bank of English (Collins and U Birmingham) has more than 500 million words
Available in many languages
Easy to collect: The web is the largest corpus ever built and within the reach of a mouse click
Parallel: same text in two languages: English/French (Canadian Hansards), European parliament (23 languages)
Annotated with part-of-speech or manually parsed (treebanks):

- Characteristics/N of/PREP Current/ADJ Corpora/N
- (NP (NP Characteristics) (PP of (NP Current Corpora)))
Writing dictionaries
Dictionaries for language learners should be built on real usage

- *They’re just trying to score brownie points with politicians*
- *The boss is pleased – that’s another brownie point*

Bank of English: *brownie point* (6 occs) *brownie points* (76 occs)

Extensive use of corpora to:

- Find *concordances* and cite real examples
- Extract *collocations* and describe frequent pairs of words
Concordances

A word and its context:

<table>
<thead>
<tr>
<th>Language</th>
<th>Concordances</th>
</tr>
</thead>
<tbody>
<tr>
<td>English</td>
<td>s beginning of miracles did Je n they saw the miracles which n can do these miracles that t ain the second miracle that Je e they saw his miracles which</td>
</tr>
<tr>
<td>French</td>
<td>le premier des miracles que fi i dirent: Quel miracle nous mo om, voyant les miracles qu’il peut faire ces miracles que tu s ne voyez des miracles et des</td>
</tr>
</tbody>
</table>
Word preferences: Words that occur together

<table>
<thead>
<tr>
<th></th>
<th>English</th>
<th>French</th>
<th>German</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>You say</strong></td>
<td><em>Strong tea</em></td>
<td><em>Thé fort</em></td>
<td><em>Schmales Gesicht</em></td>
</tr>
<tr>
<td></td>
<td><em>Powerful computer</em></td>
<td><em>Ordinateur puissant</em></td>
<td><em>Enge Kleidung</em></td>
</tr>
<tr>
<td><strong>You don’t say</strong></td>
<td><em>Strong computer</em></td>
<td><em>Thé puissant</em></td>
<td><em>Schmale Kleidung</em></td>
</tr>
<tr>
<td></td>
<td><em>Powerful tea</em></td>
<td><em>Ordinateur fort</em></td>
<td><em>Enges Gesicht</em></td>
</tr>
</tbody>
</table>
### Word Preferences

<table>
<thead>
<tr>
<th>Strong w</th>
<th>Powerful w</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>strong w</em></td>
<td><em>powerful w</em></td>
</tr>
<tr>
<td>161</td>
<td>0</td>
</tr>
<tr>
<td>175</td>
<td>2</td>
</tr>
<tr>
<td>106</td>
<td>0</td>
</tr>
</tbody>
</table>

...
Corpora as Knowledge Sources

Short term:
- Describe usage more accurately
- Assess tools: part-of-speech taggers, parsers.
- Learn statistical/machine learning models for speech recognition, taggers, parsers
- Derive automatically symbolic rules from annotated corpora

Longer term:
- Semantic processing
- Texts are the main repository of human knowledge
Finite-State Automata

A flexible tool to search and process text
A FSA accepts and generates strings, here \( ac, abc, abbc, abbbbc, abbbbbb... \), etc.
Mathematically defined by

- $Q$ a finite number of states;
- $\Sigma$ a finite set of symbols or characters: the input alphabet;
- $q_0$ a start state,
- $F$ a set of final states $F \subseteq Q$
- $\delta$ a transition function $Q \times \Sigma \rightarrow Q$ where $\delta(q, i)$ returns the state where the automaton moves when it is in state $q$ and consumes the input symbol $i$. 
FSA in Prolog

% The start state
start(q0).

% The final states
final(q2).

transition(q0, a, q1).
transition(q1, b, q1).
transition(q1, c, q2).

accept(Symbols) :-
    start(StartState),
    accept(Symbols, StartState).

accept([], State) :-
    final(State).
accept([Symbol | Symbols], State) :-
    transition(State, Symbol, NextState),
    accept(Symbols, NextState).
Regular Expressions

Regexes are equivalent to FSA and generally easier to use

Constant regular expressions:

<table>
<thead>
<tr>
<th>Pattern</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>regular</td>
<td>A section on regular expressions</td>
</tr>
<tr>
<td>the</td>
<td>The book of the life</td>
</tr>
</tbody>
</table>

The automaton above is described by the regex `ab*c`

grep 'ab*c' myFile1 myFile2
### Metacharacters

<table>
<thead>
<tr>
<th>Chars</th>
<th>Descriptions</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>Matches any number of occurrences of the previous character – zero or more</td>
<td>ac*e matches strings ae, ace, acce, accce, etc. as in “The aerial _acceleration alerted the ace pilot”</td>
</tr>
<tr>
<td>?</td>
<td>Matches at most one occurrence of the previous character – zero or one</td>
<td>ac?e matches ae and ace as in “The _erial acceleration alerted the ace pilot”</td>
</tr>
<tr>
<td>+</td>
<td>Matches one or more occurrences of the previous character</td>
<td>ac+e matches ace, acce, accce, etc. as in “The _erial _eleration alerted the ace pilot”</td>
</tr>
</tbody>
</table>
## Metacharacters

<table>
<thead>
<tr>
<th>Chars</th>
<th>Descriptions</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>{n}</td>
<td>Matches exactly $n$ occurrences of the previous character</td>
<td>\texttt{ac{2}e} matches \textit{acce} as in “The aerial \underline{ac}celeration alerted the \underline{ace} pilot”</td>
</tr>
<tr>
<td>{n,}</td>
<td>Matches $n$ or more occurrences of the previous character</td>
<td>\texttt{ac{2,}e} matches \textit{acce}, \textit{accce}, etc.</td>
</tr>
<tr>
<td>{n,m}</td>
<td>Matches from $n$ to $m$ occurrences of the previous character</td>
<td>\texttt{ac{2,4}e} matches \textit{acce}, \textit{accce}, and \textit{acccce}.</td>
</tr>
</tbody>
</table>

Literal values of metacharacters must be quoted using \.
The dot . is a metacharacter that matches one occurrence of any character except a new line.
a.e matches the strings ale and ace in:

*The aerial acceleration alerted the ace pilot*

as well as age, ape, are, ate, awe, axe, or aae, aAe, aBe, a1e, etc.

.* matches any string of characters until we encounter a new line.
The Longest Match

The previous slide does not tell about the match strategy. Consider the string *aabbcc* and the regular expression *a+b*

By default the match engine is greedy: It matches as early and as many characters as possible and the result is *aab*-

Sometimes a problem. Consider the regular expression `<b>..*</b>` and the phrase

*They match* `<b>`*as early*`</b>` *and* `<b>`*as many*`</b>` *characters as they can.*

It is possible to use a lazy strategy with the `*?` metacharacter instead: `<b>..*?"</b>` and have the result:

*They match* `<b>`*as early*`</b>` *and* `<b>`*as many*`</b>` *characters as they can.*
[... ] matches any character contained in the list.
[^...] matches any character not contained in the list.
[abc] means one occurrence of either a, b, or c
[^abc] means one occurrence of any character that is not an a, b, or c,
[ABCDEFGHIJKLMNOPQRSTUVWXYZ] one upper-case unaccented letter
[0123456789] means one digit.
[Cc]omputer [Ss]cience matches Computer Science, computer Science, Computer science, computer science.
## Predefined Character Classes

<table>
<thead>
<tr>
<th>Expr.</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>\d</td>
<td>Any digit. Equivalent to [0–9]</td>
<td>A\dC matches A0C, A1C, A2C, A3C etc.</td>
</tr>
<tr>
<td>\D</td>
<td>Any nondigit. Equivalent to [^0–9]</td>
<td></td>
</tr>
<tr>
<td>\w</td>
<td>Any word character: letter, digit, or underscore. Equivalent to [a-zA-Z0-9_]</td>
<td>1\w2 matches 1a2, 1A2, 1b2, 1B2, etc</td>
</tr>
<tr>
<td>\W</td>
<td>Any nonword character. Equivalent to [^\w]</td>
<td></td>
</tr>
<tr>
<td>\s</td>
<td>Any white space character: space, tabulation, new line, form feed, etc.</td>
<td></td>
</tr>
<tr>
<td>\S</td>
<td>Any nonwhite space character. Equivalent to [^\s]</td>
<td></td>
</tr>
</tbody>
</table>
### Nonprintable Symbols or Positions

<table>
<thead>
<tr>
<th>Char.</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>^</td>
<td>Matches the start of a line</td>
<td>^ab*c matches ac, abc, abbc, etc. when they are located at the beginning of a new line</td>
</tr>
<tr>
<td>$</td>
<td>Matches the end of a line</td>
<td>ab?c$ matches ac and abc when they are located at the end of a line</td>
</tr>
<tr>
<td>\b</td>
<td>Matches word boundaries</td>
<td>\babc matches abcd but not dabc</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bcd\b matches abcd but not abcde</td>
</tr>
<tr>
<td>\n</td>
<td>Matches a new line</td>
<td>a\nb matches</td>
</tr>
<tr>
<td></td>
<td></td>
<td>a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>b</td>
</tr>
<tr>
<td>\t</td>
<td>Matches a tabulation</td>
<td></td>
</tr>
</tbody>
</table>
Union and Boolean Operators

Union denoted \( \mid \): \( a \mid b \) means either \( a \) or \( b \).

Expression \( a \mid bc \) matches the strings \( a \) and \( bc \) and \( (a \mid b)c \) matches \( ac \) and \( bc \),

Order of precedence:

1. Closure and other repetition operator (highest)
2. Concatenation, line and word boundaries
3. Union (lowest)

\( abc^* \) is the set \( ab, abc, abcc, abccc, \) etc.

\( (abc)^* \) corresponds to \( abc, abcabc, abcabcabc, \) etc.
Perl

Match

```perl
while ($line = <>) {
    if ($line =~ m/ab+c/) {
        print $line;
    }
}
```

Substitute

```perl
while ($line = <>) {
    if ($line =~ m/ab+c/) {
        print "Old: ", $line;
        $line =~ s/ab+c/ABC/g;
        print "New: ", $line;
    }
}
```
**Perl**

**Translate**

```perl
tr/ABC/abc/
=line =~ tr/A-Z/a-z/;
=line =~ tr/AEIOUaeiou//d;
=line =~ tr/AEIOUaeiou/$/cs;
```

**Concatenate**

```perl
while ($line = <>) {
    $text .= $line;
}
print $text;
```

**References**

```perl
while ($line = <>) {
    while ($line =~ m/\$ *[0-9]+\.[0-9]+/g) {
        print "Dollars: ", $1, " Cents: ", $2; 
    }
}
```
Predefined variables

```perl
$line = "Tell me, O muse, of that ingenious hero
who travelled far and wide after he had sacked
the famous town of Troy.";
$line =~ m/,.*,;/
print $&, "\n";
print "Before: ", $', "\n";
print "After: ", $', "\n";
```

Arrays

```perl
@array = (1, 2, 3); #Array containing 1, 2, and 3
print $array[1]; #Prints 2
```
Concordances in Perl

# Modified from Doug Cooper
($file_name, $string, $width) = @ARGV;
open(FILE, "$file_name")
    || die "Could not open file $file_name."
while ($line = <FILE>) {
    $text .= $line;
}
$string =~ s/ /\s/g; # spaces match tabs and new lines
$text =~ s/\n/ /g; # new lines are replaced by spaces
while ($text =~ m/(.{0,$width}$string.{0,$width})/g ) {
    # matches the pattern with 0..width to the right and left
    print "$1
"; #$1 contains the match
}

Pierre Nugues
Language Processing with Perl and Prolog
Approximate String Matching

A set of edit operations that transforms a source string into a target string: copy, substitution, insertion, deletion, reversal (or transposition). Edits for *acress* from Kernighan et al. (1990).

<table>
<thead>
<tr>
<th>Typo</th>
<th>Correction</th>
<th>Source</th>
<th>Target</th>
<th>Position</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>across</td>
<td>actress</td>
<td>–</td>
<td>t</td>
<td>2</td>
<td>Deletion</td>
</tr>
<tr>
<td>across</td>
<td>cress</td>
<td>a</td>
<td>–</td>
<td>0</td>
<td>Insertion</td>
</tr>
<tr>
<td>across</td>
<td>caress</td>
<td>ac</td>
<td>ca</td>
<td>0</td>
<td>Transposition</td>
</tr>
<tr>
<td>across</td>
<td>access</td>
<td>r</td>
<td>c</td>
<td>2</td>
<td>Substitution</td>
</tr>
<tr>
<td>across</td>
<td>across</td>
<td>e</td>
<td>o</td>
<td>3</td>
<td>Substitution</td>
</tr>
<tr>
<td>across</td>
<td>acres</td>
<td>s</td>
<td>–</td>
<td>4</td>
<td>Insertion</td>
</tr>
<tr>
<td>across</td>
<td>acres</td>
<td>s</td>
<td>–</td>
<td>5</td>
<td>Insertion</td>
</tr>
</tbody>
</table>
Minimum Edit Distance

Edit distances measure the similarity between strings. We compute the minimum edit distance using a matrix where the value at position \((i,j)\) is defined by the recursive formula:

\[
edit\_distance(i,j) = \min \left( \begin{array}{c}
edit\_distance(i-1,j) + \text{del\_cost} \\
edit\_distance(i-1,j-1) + \text{subst\_cost} \\
edit\_distance(i,j-1) + \text{ins\_cost}
\end{array} \right).
\]

where \(edit\_distance(i,0) = i\) and \(edit\_distance(0,j) = j\).
Edit Operations

\[
\begin{align*}
(i-1, j) & \xrightarrow{\text{delete}} (i, j) \\
(i-1, j-1) & \xrightarrow{\text{replace}} (i, j) \\
i, j-1 & \xrightarrow{\text{insert}} (i, j)
\end{align*}
\]

Usually, \( \text{del\_cost} = \text{ins\_cost} = 1 \)

\( \text{subst\_cost} = 2 \) if \( \text{source}(i) \neq \text{target}(j) \)

\( \text{subst\_cost} = 0 \) if \( \text{source}(i) = \text{target}(j) \).
Distance between \( ab \) and \( cb \)

Let us align

<table>
<thead>
<tr>
<th>Source</th>
<th>c</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>Destination</td>
<td>c</td>
<td>b</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Start</td>
<td>a</td>
<td>b</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
&i - 1, j - 1 &\text{delete} &i, j \\
&i - 1, j - 1 &\text{replace} &i, j \\
&i - 1, j &\text{insert} &i, j
\end{align*}
\]
Distance between \( ab \) and \( cb \)

Let us align \( \begin{array}{cc} a & b \\ c & b \end{array} \) Source \( \begin{array}{c} \text{Start} \\ 0 \ 1 \ 2 \end{array} \) Destination

\[
\begin{array}{ccc}
    & b & 2 \\
  c & 1 & 2 \\
  \text{Start} & 0 & 1 & 2 \\
\end{array}
\]

Start \( a \) \( b \)
Distance between $ab$ and $cb$

Let us align

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>b</td>
</tr>
</tbody>
</table>

Destination

<table>
<thead>
<tr>
<th>b</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Start</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Start a b</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table showing the alignment and costs:

- Insertion: $i - 1, j - 1$ -> $i, j$
- Delete: $i - 1, j$ -> $i, j$
- Replace: $i - 1, j - 1$ -> $i, j - 1$

Note: The table shows the costs for aligning the sequences.
Distance between *ab* and *cb*

Let us align

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>Source</td>
</tr>
<tr>
<td>c</td>
<td>b</td>
<td>Destination</td>
</tr>
</tbody>
</table>

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>c</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Start</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Start</td>
<td>a</td>
<td>b</td>
<td></td>
</tr>
</tbody>
</table>

Diagram:

- **Delete**: $i-1,j \rightarrow i,j$
- **Replace**: $i-1,j-1 \rightarrow i,j-1$
- **Insert**: $i,j \rightarrow i,j+1$
Distance between *language* and *lineage*

<table>
<thead>
<tr>
<th></th>
<th>e</th>
<th>g</th>
<th>a</th>
<th>e</th>
<th>n</th>
<th>i</th>
<th>l</th>
<th>Start</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>Language</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Start  l  a  n  g  u  a  g  e
Distance between *language* and *lineage*

<table>
<thead>
<tr>
<th></th>
<th>7</th>
<th>6</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>g</td>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>a</td>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>e</td>
<td>4</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>n</td>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>i</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>l</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Start: 0 1 2 3 4 5 6 7 8

Start: l a n g u a g e
Distance between *language* and *lineage*

<table>
<thead>
<tr>
<th></th>
<th>e</th>
<th>g</th>
<th>a</th>
<th>e</th>
<th>n</th>
<th>i</th>
<th>l</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>6</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>g</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>a</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>e</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>n</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>i</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>l</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

Start: 0 1 2 3 4 5 6 7 8

Start language
Perl Code

```perl
($source, $target) = @ARGV;
$length_s = length($source);
$length_t = length($target);
# Initialize first row and column
for ($i = 0; $i <= $length_s; $i++) {
    $table[$i][0] = $i;
}
for ($j = 0; $j <= $length_t; $j++) {
    $table[0][$j] = $j;
}
# Get the characters. Start index is 0
@source = split(//, $source);
@target = split(//, $target);
```
# Fills the table. Start index of rows and columns is 1
for ($i = 1; $i <= $length_s; $i++) {
    for ($j = 1; $j <= $length_t; $j++) {
        # Is it a copy or a substitution?
        $cost = ($source[$i-1] eq $target[$j-1]) ? 0 : 2;
        # Computes the minimum
        $min = $table[$i-1][$j-1] + $cost;
        if ($min > $table[$i][$j-1] + 1) {
            $min = $table[$i][$j-1] + 1;
        }
        if ($min > $table[$i-1][$j] + 1) {
            $min = $table[$i-1][$j] + 1;
        }
        $table[$i][$j] = $min;
    }
}
print "Minimum distance: ", $table[$length_s][$length_t], "\n";
% edit_operation carries out one edit operation
% between a source string and a target string.
edit_operation([Char | Source], [Char | Target], Source, Target, ident, 0).
edit_operation([SChar | Source], [TChar | Target], Source, Target, sub(SChar,TChar), 2) :-
   SChar \= TChar.
edit_operation([SChar | Source], Target, Source, Target, del(SChar), 1).
edit_operation(Source, [TChar | Target], Source, Target, ins(TChar), 1).
% edit_distance(+Source, +Target, -Edits, ?Cost).
edit_distance(Source, Target, Edits, Cost) :-
    edit_distance(Source, Target, Edits, 0, Cost).

edit_distance([], [], [], Cost, Cost).
edit_distance(Source, Target, [EditOp | Edits], Cost, FinalCost) :-
    edit_operation(Source, Target, NewSource, NewTarget, EditOp, CostOp),
    Cost1 is Cost + CostOp,
    edit_distance(NewSource, NewTarget, Edits, Cost1, FinalCost).
Distance between *language* and *lineage*

<table>
<thead>
<tr>
<th></th>
<th>First alignment</th>
<th>Third alignment</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Without epsilon symbols</strong></td>
<td><code>language</code></td>
<td><code>language</code></td>
</tr>
<tr>
<td></td>
<td><code>lineage</code></td>
<td><code>lineage</code></td>
</tr>
<tr>
<td><strong>With epsilon symbols</strong></td>
<td><code>language</code></td>
<td><code>language</code></td>
</tr>
<tr>
<td></td>
<td><code>lineageε</code></td>
<td><code>lineageε</code></td>
</tr>
</tbody>
</table>