Wmatch - User Manual

Sophie Rosset & Olivier Galibert

Contents

1 Wmatch

1.1 Introduction

Wmatch is and engine that matches (and substitutes) regular expressions using words as the base unit instead of characters. This property allows for a more readable syntax than traditional regular expressions and enables the use of classes (lists of words) and macros (sub-expressions in-line in a larger expression). `Wmatch` includes also NLP-oriented features like strategies for prioritizing rule application, recursive substitution modes, word tagging (for tags like noun, verb...), word categories (number, acronym, proper name...). It has multiple input and output formats, including an XML-based one for interoperability and to allow chaining of instances of the tool with different rule sets. Rules are pre-analyzed and optimized in several ways, and stored in compact format in order to speed up the process. Analysis is multi-pass, and subsequent rule applications operate on the results of previous rule applications which can be enriched or modified.

Wmatch includes:

- positive and negative lookaheads.
- shy and greedy groupings.
- names classes and macros.
- strategies for prioritizing the rules application.
- tree-creating substitutions, and search in sub-trees.
- external word tagging (for example for semantic or morphosyntactic classes).
- internal words categorization. (number, acronyms, etc.)
- regular expression on characters within a word.

Here is a simple example of regular expression on word:

_loc: ((<= &prep_loc) %caps | %country | %city);

Where _loc is the name of the rule (the entity type that we want to match), &prep_loc is referring to a macro that describes preposition and prepositional phrase, all designing a location, %caps the class of words beginning with an upper-case, and %country and %city the classes of words belonging to a lexicon of countries and cities.

1.2 Rule file format

General Format

```
pattern-file:
        pattern-file class-definition
|       pattern-file macro-definition
|       pattern-file rule-definition
|       // empty
;
```

The file is a succession of definitions of classes, macros and rules, and also instruction to the pre-processor. The order is not important.
The file is first given to the pre-processor C to allow the use of includes, define, etc. The C/C++ comments, ie /* ... */ et // ... are recognized as such.

Lexical level

A word is defined as being a string between two special characters or spaces. The special characters are:

```
\()*^$+?,"[]{}.;:|.
```

If a special character is part of a word, the word has to be put between '"'. For example, the name *O'Connell* contains the spacial character "'" and when used in a rules or in a lexicon, it has to be put as **"O'Connell"**.
The end of line character is a blank character with no special meaning.

A quotation mark '"' introduces a word that must end with another '"' and before the end of the line. If it is to contain a'"', it must be written \".

A word begining by '%' is a class-name. A word begining by '&' is a macro-name. The remaining are simple word.

Classes

```
class-definition:
        class-name ':' class-list ';'
;

class-list:
        class-list? word
|       class-list? class-name
;
```

A class is a list of words that one want to match, such as “one two three”. A class can also include other classes that can be defined elsewhere in the file. Circular inclusion are forbidden. A class can not be void.

The predefined classes are:

- %caps : words having one uppercase at the beginning; moreover the word can have one uppercase inside the word if this uppercase is preceded by a dash (“-”); the other letters are lowercases. For example *Paris* and *tata-Bidule* belongs to the %cap class.
- %acronym : words without any lowercase. For example *NATO* belongs to the class %acronym.
- %number() : numbers (with the possibility to define interval).
- %cardinal() : cardinals (with the possibility to define interval).
- %ordinal() : ordinals (with the possibility to define interval).
- %tag : tags defined by the analyzer (the grammars). In the first example, all the nodes beglon the the %tag class (only if they begin, following the defined convention, by “_”).
- %tagm : words begining with “_ ”.
- %nonalpha : words containing at least a character different from uppercase, lowercase, number and “”.
- %alpha : words having no uppercase.
- %mixed : other.

Macros

A macro is a regular sub-expression which can be used in other expression definitions and rule definition. Circular inclusion are forbidden. All the macro are defined as follows:

```
macro-definition:
        macro-name ':' regexp ';'
;
```

Les Règles

```
rule-definition:
        substitution-list '{' word '}' ':' regexp ';'   // Regle nommee
|       substitution-list              ':' regexp ';'   // Regle anonyme
;
```

```
substitution-list:
        substitution-list? word
;
```

A rule is a list of substitution names to which the matched zones of the regular expression will be associated and then the expression itself. It is possible to give it a name. There is no need to give unique name.

Regular Expression

Wmatch includes standard regular expression syntax features:

```
regexp:
        /* empty */                          // Expression vide
|       word                                 // Simple word
|       '.'                                  // Any word
|       class-name                           // Predefined class
|       '[' '^'? class-list ']'              // Inline defined class, can
                                                ... be negated
|       '^'                                  // Begining of text
|       '$'                                  // End of text
|       macro-name                           // Macro
|       regexp regexp                        // Consecutive matching
|       regexp '*'                           // 0-infinity matching
|       regexp '+'                           // 1-infinity matching
|       regexp '?'                           // 0-1 matching
|       regexp '{' word? ',' word? '}'       // n-m matching (the words must
                                                ... be numbers)
|       regexp '{' word '}'                  // n matching
|       regexp '*' '?'                       // 0-infinity matching shy
|       regexp '+' '?'                       // 1-infinity matching shy
|       regexp '?' '?'                       // 0-1 matching shy
|       regexp '{' word? ',' word? '}' '?'   // n-m matching shy
|       regexp '{' word '}' '?'              // n matching shy
|       regexp '|' regexp                    // Alternative
|       '(?=' regexp ')'                     // lookahead positif droite
|       '(?!' regexp ')'                     // lookahead negatif droite
|       '(<=' regexp ')'                     // lookahead positif gauche
|       '(<!' regexp ')'                     // lookahead negatif gauche
|       '(?:' regexp ')'                     // non-substituting grouping
```

```
|       '('  regexp ')'                          // substituting grouping
;
```

Wmatch is a standard and modern regexp system.

- the greedy operators match as much as possible, the shy operators as less as possible.
- the priority order of matchings is alternative matching < consecutive matching < matching op, i.e. "a b | c d *" is "(a b)|(c (d *))"
- macros are equivalent to write '(?:' contenu ')' in the expression

The substitution groups define what will be retrieved in the output.

A rule is as following:

w1 w2 w3 : ... (part1) .. (part2) ... (part3) ... ;

She will associate part1 to w1, part2 to w2 and part3 to w3. The number of expression in parenthesis in the right should be the same as the number of names in left. The tree "... (... (...)) ..." is possible, as well as the repetition "... (...)+ ...", but it is easy to rapidly loose the control of the result. So in order to use a group whithout substitution, for example to use "l", one could use \(et \).

Usage

```
Usage: wmatch [options] pattern-file [input file...]
-Idir           path include to give to cpp
-Dx[=y]         define to give to cpp
-P              don't pass the pattern file through cpp
-F<format>      input format, default alt
-T<format>      output format, default match
-w<number>      match for that word <number> is the first word of
                the first substitution
-o<file>        output file (otherwise stdout)
-f<file>        list of words to filter out from the output
-s              substitution mode
-S              global substitution mode
-r              repeat mode
```

```
-R              left-recursion mode (incompatible with -s and -r and -N)
-N              right-recursion mode (incompatible with -s and -r and -R)
-i              align matched on rightmost group
-t              Tick-mode, show times of various steps
-x              Compile2 test mode
-p              Macro/rules profiling

Input  formats: txt txtlines alt altlines xml xtag xtaglines html
Output formats: match group cases tag xml scframes xtag raw
```

2 Examples WMATCH

2.1 Resources

- Classes:
  - %weekday: Monday Tuesday Wednesday Thursday Friday Saturday Sunday;
  - %month: January Fevruary March April ...;
- Macros:
  - &firstname: Jean Paul | Jean | Pierre ;

It is possible (and actually the best thing to do) to use external resources. With these external resources (on the wmatch format file) you can do *includes* within the rules file.

For example, the file fname.lst contains two macros:

```
[149 rosset@toto ~/nca/listes]$ cat prenoms.lst
&pren1 :
Aadel |
Aadhawen |
Aadil |
Aadnan |
Aage |
Aahad |
Aahah |
...
;
&pren2 :
Aabye |
Aadne |
Aall |
```

```
Aaron |
Abbot |
Abbott |
...
;

[149 rosset@toto ~/nca/listes]$ cat pers.wm
#include fname.lst

_prenom: (&pren1 | &pren2);
```

In the following example, different lexicons are included in one general macro in the rule file:

```
[149 rosset@m23 ~/nca/listes]$ cat fname1.lst
&f1 :
Aadel |
Aadhawen |
Aadil |
Aadnan |
Aage |
Aahad |
Aahah |
...
;

[149 rosset@m23 ~/nca/listes]$ cat fname2.lst
&f2 :
Aabye |
Aadne |
Aall |
Aaron |
Abbot |
Abbott |
...
;

[149 rosset@m23 ~/nca/listes]$ cat pers.wm
&fname:
#include fname1.lst
|
#include fname2.lst
;
```

```
_firsname: (&fname);
```

2.2 Rules

- Simple rules:
  - _date: (%weekday %number(1,31) %month);
  - _pers: (&fname %caps);
- Multiple rules:
  - _fname _lname: (&fname) (%caps (?! %caps));
  - _week _day _month: (%weekday) (%number(1,31)) (%month);
- And finally grouping in one node in a next step (grammar):
  - _pers: (_fname _name);
  - _date: (_week _day _month);

2.3 Tree manipulation

The output of a wmatch analysis is a tree. Each pass add, delete, modify etc. nodes in these trees. The operators to walk through the trees explicitely (if one want work on the tree structure and not only on the upper nodes) are:

2.3.1 Simple match

« ... » : match every word at the same level
<* ... *> : match everywhere
<+ ... +> : match nodes without any child

2.3.2 Possible operators

There are different operators: delete, rename, replace

Word Deletion
%delete_word(%1): (_TOTO) « w1 w2 ... »;

→ %1 that is to be deleted is between (). After the ")", the conext of the deletion is indicated. One could have:

%delete_word(%1): (_TOTO);

that will delete every tag _TOTO

Word Replacement
%replace_word(%1, _TOTO): (_TATA);

→ replace %1 with _TOTO. %1 is in parenthesis (_TATA in this example). A context can be defined:

%replace_word(%1, _TOTO): A (_TATA);
Tree replacement:
%replace_tree(%1, _TOTO « %2 »): (_TATA « _TITI « (.+) » »);

%1 correspond to the first parenthesis and %2 to the second one. Here, the tree _TATA « _TITI « blabla » » is replaced with _TOTO « blabla ».

2.4 Wmatch in action

2.4.1 Rules and simple include

```
#include "utils.wm"

&country: #include "country.lst";
&city: #include "city.lst";

&prep_loc: (<= &vbe_move) &prep;

_loc: ((<= &prep_loc) %caps | &country | &city);
```

where :

- **_loc** = name of the rule
- **&prep_loc** = macro that define prepositions or prepositional phrases placed before location.
- **&country** et **&city** = macro calling for a lexicon of countries or cities.

2.4.2 Tree manipulation

Simple Match

let's consider the following analysis:
<_det> the </_det> <_function> president </_function> <_prep> of </_prep> <_country> France </_country>

Objective: one want tag a _country as an _org if and only if it is preceded by a function that contains *president* and possibly followed by a preposition or a determinant.

%prepdet: _prep _det;

_org: ((<= _function « president » %prepdet?) _country);

Deletion
%delete_word(%1): (_function) « was the head of »;

→ %1 that has to be deleted is in parenthesis. Here the node _function has to be deleted when it contains *was the head of*. One could have:

%delete_word(%1): (_function);

that would delete every node _function.

Word replacement
%replace_word(%1, _Rfunc.adm): (_function_adm);

→ replace the %1 with _Rfunc.adm. %1 is indicated in parenthesis. It is _function_adm. One could have defined a context:

%replace_word(%1, _Rfunc.adm): A (_function_adm);
→ the replacement is done only if the node _function_adm is preceded with *A*.

Tree replacement: example 1
Departure analysis: <_loc> <_phone> 01 69 85 80 02 </_phone> </_loc>

Objective: <_Rloc.addr.phone> 01 69 85 80 02 </_Rloc.addr.phone>

Do:

%replace_tree(%1, _Rloc_addr.phone « %2 »): (_loc « _phone « (.+) » »);

%1 corresponds to the first parenthesis; %2 to the second one. Here:

%1 = the complete group _loc is replaced with _Rloc.addr.phone
%2 = what is in the _phone node.

Tree replacement: example 2
Departure analysis for *10 12 hours*:
<_time> <_year> 10 </_year> </_time>
<_val_unitTime> <_val> 12 </_val> <_unitTime> hours </_unitTime>
</_val_unitTime>

Objective: <_duration> <_val> 10 </_val> <_val> 12 </_val> <_unit> hours </_unit> </_duration>

Do:

%h: hours minutes mn hrs;
%replace_tree(%1, _duration « _val « %2 » %3 _unit « %4 » »): (_time « _year « (%cardinal()) » » _val_unitTime « (_val « %cardinal() ») _unitTime « (%h) »);

→ replace %1 (from _time to _val_unitTime) with _duration which contains a _val with %2 (a cardinal, here *10*) and the %3 which already contains a _val including a cardinal.

3 Packaging of WMATCH

Every grammar are described in a language based on Lua[1].

[1] `www.lua.org`

3.1 Example of a script

```
#!/PATH/TO/WMATCH/wmatch

h = "/PATH/TO/WORKDIR"

paths.rules =  h.."/rules", h.."/lists"
paths.dicos =  h.."/dic"
paths.filters =  h.."/filters"

cache("/PATH/TO/CACHE/my-cache")

r = input()

--pre-analysis on character level
r = word_tag_full(r, "my-character-level-grammar.wm")

--one standard grammar on word level
r = match_global_replace(r, "my-standard-grammar.wm")

(...)

--filtering non-necessary tags
r = filter(r, "my-filter")

output(r)
```

3.2 Input and Output Formats

The default formats are txtlines for input and xtag for output. If different script (each represeting a grammar) are chained, then from the second one to the last one the input format should be xtaglines.

Command to call your wmatch system on a document:

fr-time.lua -Fxtaglines -Txtag < my-text.in > my-text.out

Also in an interactive mode:

fr-time.lua -Fxtaglines -Txtag <RETURN>

Type your text