Word List Markup Language (WLML)

EBFE
ebfe@inbox.ru

10. Juli 2009

Inhaltsverzeichnis

Einführung

Vorwort

WLML ist eine Domänensprache und eignet sich nur für einen kleinen Aufgabenbereich. Dafür wird man diese Aufgaben mit paar Zeilen Code erledigen können - anstatt mehrere Seiten in einer universellen Sprache wie C/C++/VB/Java/Delphi usw. zu tippen.

Zweck

Mit WLML lassen sich bestimmte „Worteigenschaften“ beschreiben und sich Anhand der Beschreibung komplette Wortlisten generieren. Oder bestehende Wortlisten modifizieren. Es kann überall dort eingesetzt werden, wo man fein abgestufte Wörterbücher braucht oder kombinatorische Aufgaben lösen muss. Das können zum Beispiel vergessene RAR/ZIP/TrueCryptVolumes Passwörter sein, von denen man noch Teile kennt.

Also alle Situationen, bei denen reine Brute-Force Angriffe langsam sind (5 bis 500 Wörter pro Sekunde) und man mit intelligenten Listen viel Zeit sparen kann. Oder wenn man mehrere Wörter/Zeichen nach komplexen Regeln kombinieren möchte (zum Beispiel **alle möglichen** E-Mail-Adressen aus Vorname/Name/Internetdomain generieren). Siehe auch Beispiele.

Grundlegende Konzepte

In klassischen imperativen Sprachen wie C/C++/Pascal/Basic muss man mit einem Algorithmus dem Computer beibringen *wie* er zu einem Ergebnis kommt. WLML ist dagegen deklarativ - man beschreibt also *was* man haben möchte - wie das Ergebnis nun erreicht wird, ist die Sorge des Rechners (dafür ist er ja da ☺).

Da die Sprache leicht erlernbar sein soll, wurden die Schlüsselwörter und Operatoren mehreren Sprachen entlehnt. Fast immer gibt es für eine Eingabe mehrere Alternativen. Wer Semikolon(;) aus Gewohnheit an das Zeilenende setzt, braucht sich auch nicht umzugewöhnen. Zeilenumbrüche, Leerzeichen und Tabulatoren werden ebenfalls übersprungen.

Im Moment werden WLML Dateien in Zwischencode übersetzt, optimiert und interpretiert. Der Interpreter wurde in Prolog geschrieben.

Installation und Start des Interpreters

Windows

Archiv herunterladen und entpacken. Nun gibt es mehrere Alternativen:

Installation(empfohlen): Im `windows` Unterordner `install.cmd` starten. Sie verknüpft die Dateiendung `.wlml` mit dem Interpreter, so dass man die Dateien direkt per Doppelklick starten kann. Mit `uninstall.cmd` wird die Verknüpfung wieder entfernt.

nur Interpreterstart: Die `start.cmd` im Hauptordner startet die Interpreterkonsole.

selbst compilieren: Wer es unbedingt selbst compilieren möchte, sollte SWI-Prolog von www.swi-prolog.org installieren. Nach der Installation `cmd.pl` im Hauptordner anklicken oder `startpl.cmd` starten. Eingabe von `plcon -c cmd.pl meine.wlml` in der CMD funktioniert ebenfalls.

Linux

Archiv herunterladen und entpacken. Nun wird man zuerst Prolog installieren müssen:

Suse, Mandriva: Hier kann das Ganze über den eingebauten grafischen Installer erledigt werden. Sollte `swi-prolog` nicht verfügbar sein, kann hier: www.swi-prolog.org ein RPM-Packet heruntergeladen werden.

Debian und Derivate U/K/Xbuntu, Xandros, DSL usw.:

```
apt-get update
apt-get install swi-prolog swi-prolog-*
```

Gentoo, Arch&Co: Ich gehe davon aus, dass diese Nutzer in der Lage sind, `swi-prolog` alleine zu in installieren ☺

nun kann man entweder den Interpreter starten:

```
ebfe@ubuntu:~$ prolog -c /home/ebfe/Desktop/cmd.pl
```

oder direkt eine Datei compilieren:

```
ebfe@ubuntu:~$ cd Desktop/
ebfe@ubuntu:~/Desktop$ prolog -c /home/ebfe/Desktop/cmd.pl ↩
    hello.wlml
```

Wer es bequem mag, kann `install.sh` ausführen:

```
ebfe@ubuntu:~/Desktop$ chmod +x install.sh
ebfe@ubuntu:~/Desktop$ sudo ./install.sh
```

sie erstellt ein Bashscript in `/user/local/bin/wlml`

```
#!/bin/bash
prolog --quiet -c /home/ebfe/Desktop/cmd.pl $1 [1]
```

Damit lässt sich der Interpreter über `wlml` starten und mit `wlml dateiname` der Quellcode direkt in der Konsole compilieren. Alternativ fängt man den Code mit `#!/usr/local/bin/wlml` an - dann geht auch ein `./datei.wlml` Aufruf:

[1] der Pfad wird automatisch angepasst

```
ebfe@ubuntu:~/Desktop$ more mytest.wlml
#!/usr/local/bin/wlml
define hi="abc"
result='hallo'+hi
ebfe@ubuntu:~/Desktop$ wlml mytest.wlml
Word List Markup Language (WLML) Interpreter
Autor: EBFE (ebfe at inbox.ru)
Sprache: Prolog
mytest.wlml wurde erfolgreich compiliert
|: exit
ebfe@ubuntu:~/Desktop$ ./mytest.wlml
Word List Markup Language (WLML) Interpreter
Autor: EBFE (ebfe at inbox.ru)
Sprache: Prolog
./mytest.wlml wurde erfolgreich compiliert
|: exit
ebfe@ubuntu:~/Desktop$
```

Mac OS X

Archiv herunterladen und entpacken. Von dieser Seite www.swi-prolog.org kann man Prolog beziehen. Mit

```
ebfe@nicht_vorhandenem_mac:~$ prolog -c cmd.pl
```

sollte der Interpreter gestartet werden können. Wenn mir jemand einen Mac spendiert, mache ich auch eine detailliertere Anleitung dafür. Versprochen :)

Der interaktive Modus

Interpreterkonsole

Wie schon in dieser Ausgabe zu sehen, ist die Interpreterkonsole interaktiv. Die Befehle können sowohl groß- wie auch kleingeschrieben werden. Grundsätzlich können die Pfadangaben sowohl zwischen ' wie auch " stehen. Folgende Kommandos stehen zur Verfügung (Alternativen sind durch Kommata getrennt):

help, ?, -?: Zeigt die Hilfe an.

quit, q, exit, e: Beendet den Interpreter.

compile, c 'Dateiname':

compile, c "Dateiname": Compiliert WLML-Datei.

test,t <NUM>: Generiert testweise `<NUM>` Wörter und gibt sie aus. `<NUM>` ist optional - ohne die Angabe werden 5 Wörter generiert.

genlist, g 'Dateiname':

genlist, g "Dateiname": Generiert Wörterliste.

Compilierung und Generierung

Vorher wurde schon beschrieben, wie man unter Windows/Linux WLML Dateien direkt mit dem Interpreter verknüpfen kann. Damit wird eine Datei direkt beim Öffnen compiliert. Ansonsten gibt man in der Interpreterkonsole

`compile "datei.wlml"` ein. Dabei wird der Code auch geprüft und optimiert, bei vorhanden Fehlern eine Meldung ausgegeben. Die häufigsten Fehler sollten zwar abgedeckt sein, den Komfort von VB.NET/C#/Java&Co Compilern darf man aber nicht erwarten :). Gegebenenfalls gibt der Interpreter einfach die Zeile aus, ab der er nicht mehr weiterkommt.

Nach der erfolgreichen Compilierung sollte das Ergebnis nochmal mit `test` getestet werden. Es werden mehrere Wörter generiert, so dass man erstmal schauen kann, ob auch das herauskommt, was man haben möchte. Zum Schluss schreibt man die komplette Liste mit `genlist "mywordlist.txt"` eine Datei. Komplettes Beispiel[2]:

```
|:compile "hello.wlml"
hello.wlml wurde erfolgreich compiliert
|:test
hi1
hi2
hi3
hi4
hi5
|:genlist "mywordlist.txt"
550 Woerter generiert in 0.01 Sekunden
```

[2]fast alle nachfolgenden Beispiele sind im `examples` Unterordner zu finden

WLML - die Sprache

Grundstruktur/Überblick

Das Wichtigste ist: in WLML *charakterisiert* man die Eigenschaften des Wortes. Man sollte also nicht wie in den „üblichen“ Sprachen versuchen, Anweisungen zu schreiben. Streng genommen weist man mit `=` auch keine Werte zu, sondern beschreibt, dass die Variable und der Ausdruck ein und dasselbe sein sollen. Eine mögliche Strukturierung sieht so aus:

Header: Includedateien, eventuell ein `#!/usr/local/bin/wlml`

Body: Variablendefition, Verknüpfungen

Result: Ergebnisdefinition

Das Einzige, was *immer* vorkommen muss, ist die `result` Definition:

```
result="abc"
```

Damit sagt man dem Interpreter, was das eigentliche Ergebnis sein soll. Vergleichbar mit `main` Definition, die in vielen anderen Sprachen verwendet wird. Der Rest ist optional und dient der besseren Lesbarkeit. So werden zum Beispiel im ersten Optimierungsschritt die die Variablen durch die Ausdrücke ersetzt:

```
define var1="abc"
define var2=var1+'123'
result=var2
#ist intern exakt dasselbe wie
result="abc"+'123'
```

Alles in einem Block zu definieren bringt also keinerlei Performancevorteile, sondern nur eine schlechtere Lesbarkeit.

Kommentare

Kommentare fangen mit `//` oder `#` an und gelten bis zum Ende der Zeile:

```
#!/usr/local/bin/wlml
define var1="abc" //var1 ist a oder b oder c
define var2='abc' #var2 ist Konstante abc
result=var2|var1
```

Variablen

Variablen werden in WLML mit `define` eingeleitet. Der Name der Variable darf aus Buchstaben, Dezimalziffern, Unterstrichen und Bindestrichen (-) bestehen. Die Gleichsetzung des Ausdrucks kann entweder schon bei der Deklaration erfolgen oder erst später im Code:

```
define word_1
define word2='word'
define a-d="abcd"
word_1=word2+"123"+a-d
```

Includedateien

Mit `include Dateiname` werden weitere Quellcodedateien eingebunden. Dateinamen können sowohl zwischen ' wie auch " stehen:

```
#!/usr/local/bin/wlml
include "myincs.inc"
include 'myincs.inc'
include ("myincs.inc")
#verwende Definitionen aus der
#Includedatei:
result=digits+a-z
```

`myincs.inc`

```
define A-Z="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
define a-z="abcdefghijklmnopqrstuvwxyz"
define digits= "1234567890"
```

Datentypen

Streng genommen gibt es nur `'Konstanten'`. Man beschreibt praktisch nur mit Hilfe dieser Einheiten und Operatoren das gewünschte Ergebnis:

```
define word1='hallo' | 'hi'     // entweder hallo oder hi
define word2='?' | '!' |'.'     // entweder ? oder ! oder .
//Ergebnis: alle Möglichkeiten von word1 verbunden mit
//allen Möglichkeiten von word2
result=word1+word2
```

Damit man nicht so viel Tippen muss, gibt es auch noch den Datentyp `"Zeichenliste"`. Aus der Liste wird immer nur ein einzelnes Zeichen genommen - es ist nur eine Kurzschreibweise für die Verknüpfung mehrerer Konstanten:

```
#das Beispiel von vorhin, ergänzt:
define word1='hallo' | 'hi'     // entweder hallo oder hi
define word2="?!."      // entweder ? oder ! oder .
define digit="123789" //entweder eine 1,2,3,7,8 oder 9
result=word1+word2+digit
```

Gekürzte Ausgabe:

```
|:test 25
hallo?1
...
hallo?7
...
hallo!1
hallo!2
...
hi?1
...
hi?7
```

Damit man ' in Konstanten oder " in Zeichenlisten aufnehmen kann, gibt es Escape-Sequenzen \' und \":

```
define word1='Max\'s passwort'
define word2="a\"c"  //a oder " oder c
result=word1+word2
```

Operatoren

Der eigentliche Kern der Sprache. Damit man sich nicht allzu sehr umgewöhnen muss, gibt es immer mehrere alternative Schreibweisen.

Auswahl

`| OR or` wählt aus zwei Blöcken einen aus:

```
define word1='Hello' | 'Hi'
define word2='Bye' or 'Cheers'
result=word1 OR word2 | 'Wtf?'
```

Ausgabe:

```
|: test
Hello
Hi
Bye
Cheers
Wtf?
```

Vereinigung

`+ & . AND and` verbinden zwei Elemente zu einem:

```
define word1='hello'
define word2='world'
define mid="- "
define num="01"
define end="?.!"
result=num + word1 & mid . word2 AND end
```

Gekürzte Ausgabe:

```
|: test 100
0hello-world?
0hello-world.
...
1hello world.
1hello world!
```

Permutation

`{Element,Element,...,Element}` bildet alle möglichen Anordnungen der Elemente. Das, was zwischen den Kommata steht, wird als Block betrachtet und nur dieser komplette Block wird getauscht. Zwei Beispiele:

```
result={'1','2','3'}
```

Ausgabe:

```
|:test
123
213
231
132
312
```

```
define char='H'|'h'
define word='ey'|'i'
define world='World'|'world'
define phrase={char+word,world}
result=phrase+'!'
```

Ausgabe:

```
|:test
HeyWorld!
WorldHey!
Heyworld!
worldHey!
HiWorld!
...
```

Optionen

Die optionalen Angaben stehen, durch Kommata getrennt, zwischen den eckigen Klammern: `[min..max,?,lowercase,uppercase,mixedcase ]`

min..max: wiederhole Element mindestens `min` und höchstens `max` Male

?: Kurzschreibweise für `[0..1]`. Element ist damit optional (kann entweder einmal oder gar nicht vorkommen)

lowercase, lower case: nur Kleinbuchstaben

uppercase, upper case: nur Großbuchstaben

mixedcase, mixed case: es sollen alle Groß/Kleinschreibungmöglichkeiten ausprobiert werden

Beispiele:

```
define chars="abc"[1..3]
result=chars[mixedcase]
```

Ausgabe:

```
|:test 100
a
A
b
B
c
C
aa
aA
Aa
AA
ab
aB
Ab
AB
...
```

```
define char="abc"
define word=char[5..5]
define mid="-_"
//mitte ist optional
result=word[mixedcase]+mid[?]+char
```

Ausgabe:

```
|:test 100
aaaaaa
aaaaab
aaaaac
...
aaaAa_a
```

```
aaaAa_b
aaaAa_c
...
aaAaA-c
aaAaA_a
aaAaA_b
```

```
define hello='Hello'
define digit="42"
result=hello[lowercase]+digit[0..2]
```

Ausgabe:

```
|:test 10
hello
hello4
hello2
hello44
hello42
hello24
hello22
```

Gruppierung

( ) fassen mehrere Elemente und Operatoren zusammen. Wie in mathematischen Ausdrücken werden zuerst die Operatoren innerhalb der Klammern angewendet. Dasselbe gilt auch für Optionen:

```
result=(('Hi'|'Hey')+' '+('world'|'Dude'))[lowercase]
```

Ausgabe:

```
|:test
hi world
hi dude
hey world
hey dude
```

```
define digits="123"
define word='hello'|'world'
result={(word|digits)[mixedcase],'!'}
```

Ausgabe:

```
|:test 1000
!World
WorlD!
...
1!
!1
2!
```

Filter

< > Elemente, die hier drin stehen, dürfen sich nicht wiederholen:

```
define digits="1234567890"
// 4 bis 5 Ziffern, die sich nicht
// wiederholen dürfen
result=<digits[4..5]>
```

Ausgabe:

```
|:test 10000
1234
1235
1236
1237
...
27805
27806
27809
```

< > werden immer auf das oberste Element/Gruppe angewendet:

```
define hello='hi'|'hello'
define chars="hier"
define chars2="he"
// hier werden nicht etwa Ergebnisse wie
// 'hellohiohl' ausgefiltert, weil da ein 'h'
// vorkommt, sondern nur wenn chars=chars2 usw. ist
// also die "obersten" Elemente
result=<hello+chars+chars2>
```

Ausgabe:

```
|: test 10
hihe
hiih
hiie
hieh
hirh
hire
hellohe
helloih
helloie
helloeh
```

Da erst das Endergebnis gefiltert wird, ist der Filter relativ langsam und sollte sparsam eingesetzt werden.

Externe Listen

Mit `file Dateiname` können externe Wörterbücher oder Listen eingebunden werden. Dateinamen können sowohl zwischen ' wie auch " stehen:

```
define names=file 'names.txt'
define lastnames=file("lastnames.txt")
result=names+lastnames
```

names.txt

```
Max
Dieter
Detlef
```

lastnames.txt

```
Müller
Schulz
Mustermann
```

Ausgabe:

```
|: test 10
MaxMüller
MaxSchulz
```

```
MaxMustermann
DieterMüller
DieterSchulz
DieterMustermann
DetlefMüller
DetlefSchulz
DetlefMustermann
```

Standarddefinitionen

Die beigelegte `std.inc` bietet Variablen A-Z, a-z und digits. Das wären Zeichen von A bis Z in groß/klein und Ziffern 1 bis 9. Ansprechbar über

```
include "std.inc"
define myword=A–Z[1..4]
password=myword+'ichbinfest'+digits
```

Beispiele

Einige erweiterte Beispiele[3]

Vergessenes Passwort I

Passwort von einem RAR Archiv war irgendwas mit `FreE-hAck`, allerdings ist inzwischen unklar, ob der Bindestrich wirklich vorhanden war und natürlich kennt man die genaue Groß/Kleinschreibung nicht mehr:

```
#RAR Passwort mit FH:
#Muster: FrEe-hAcK oder fReEHaCk
#(zufällige groß/kleinschreibung,
#optionaler bindestrich)
include ('std.inc')
define free='free'
define bind='-'[?]  //alternative zu [0..1]
define hack='hack'
result=(free+bind+hack)[mixedcase]
```

alternativ:

```
include ('std.inc')
define variante1='free-hack'[mixedcase]
define variante2='freehack' [mixedcase]
result=variante1|variante2
```

oder:

```
include ('std.inc')
result=("f"|"F")+("r"|"R")+("e"|"E")
+'usw.keine Lust komplett auszuschreiben :) '
```

[3] siehe auch Unterordner `adv_examples`

oder:

```
include ('std.inc')
password="fF"+"rR"+"eE"+"eE"+"-"[0..1]+"Hh"+"aA"+"cC"+"kK"
```

Vergessenes Passwort II

Das Passwort besteht aus 7 Zeichen, man weiß noch, dass es Buchstaben `a,b,c,d,e,f,g,h` sind. Allerdings gibt es keine Wiederholungen im Passwort - also keine `aabcd` oder `abbbcd` und ähnliche Wörter. In einem einfachen Brute Force Tool wären das $8^7 = 2097152$ Passwörter. In WLML kann man es genau beschreiben:

```
#passwort hat länge 7 und besteht aus 8
#unterschiedlichen Zeichen
#allerdings ohne Widerholungen
define zeichen="abcdefgh"
result=<zeichen[7..7]>
#alternativ (viel schneller, aber syntaktisch
#nicht so gut lesbar)
#result=<{"ah","bh","ch","dh","eh","fh","gh"}>
```

Ausgabe:

```
|:genlist "list.txt"
40320 Woerter generiert in 7.05 Sekunden
```

Somit hat man nur noch ≈ 40000 Wörter in der Liste. Bei einem TrueCrypt Volume, bei dem man nur 5 Passwörter pro Sekunde ausprobieren kann, sind es 2 Stunden Laufzeit statt 5 Tagen ☺

Lange Passwörter

```
#in einem sehr langen passwort wurde in der
#Mitte die Ziffersequenz vergessen.
#Vermutung: 4-5 Ziffern, die sich nicht widerholen
#bei einfachem BF würde man 110 000 Kombos
#durchprobieren, gegenüber 35000 (wenn man die
#"Ziffern wiederholen sich nicht" Bedingung
#miteinbringt)
```

```
include ('std.inc')
define const='ich bin ein sehr sehr langes passwort'
define varpart=<digits[4..5]>
define const2='ich bin das Ende eines langen Passworts'
password=const+varpart+const2
```

E-Mail-Adressen generieren

Das ist ein Beispiel für die erste „echte“ Anwendung der Sprache. Es geht darum, aus vorhanden Vor-, Nachnamen-, Domainlisten und einigen Bindezeichen alle möglichen E-Mail-Adressen zu generieren:

```
define nachnamen= file "namen.txt"
define vornamen= file "vornamen.txt"
define szeichen=".+-_"
define mailer= file "mailer.txt"
result = vornamen+szeichen+nachnamen+'@'+mailer
```

oder gleich:

```
define szeichen=".+-_"
result=file 'vornamen.txt'+szeichen+file 'namen.txt'+'@'+↩
    file 'mailer.txt'
```

Sonstiges

Generiert 1 bis 4 Ziffern oder 2 Buchstaben:

```
include 'std.inc' #vordefinierte Klassen wie A–Z oder ↩
    Digits (1–9)
define word1=digits[1..4]
define word2=A–Z[2..2]
password=word1|word2
```

Generiert alle Buchstaben- und Ziffernkombinationen bis zu 8 Zeichen Länge:

```
include "std.inc"
result=(A–Z|a–z|digits)[1..8]
```