# An Implementation of a Multilingual Regular Expression Segmentor for Ordinary and Morphologically Rich Lexical Tokens

Paul Wu Horng Jyh and Kevin Cheong
Institute of Systems Science, National University of Singapore, Singapore 0511
Email: <paulwu, kcheong>@iss.nus.sg

**Abstract**

Lexical pattern matching and text extraction is an essential component of many Natural Language Processing applications. Following the language hierarchy first conceived by Chomsky, it is commonly accepted that simple phrasal patterns should be categorised under the class of Regular Language (RL). There are 3 operations in RL - *Union*, *Concatenation* and *Kleene Closure* - which are applied to a finite lexicon. The machinery that recognises RL is the Finite State Machine (FSM). This paper discusses and postulates that the degree with which a class of patterns exercise the aspects of RL operators, is directly proportional to the richness in morphology of lexical tokens.

## 1 Introduction

Lexical pattern matching can be used to recognise phrasal patterns of different languages and which reflect morphological variations. For example, it is used to recognise regular English phrasal verbs like ".. give * up .." or reduplicated Chinese words like "GaoXing-Bu-GaoXing" (literally "happy-not-happy").

Simple phrasal patterns are categorised under the class of Regular Languages (RL) [2]. The degree with which lexical patterns within the language exercise the RL operators varies. Certain classes of lexical patterns, for instance (dead) idioms and Chinese lexicon items, are quite fixed. For such cases, the *Concatenation* operator in RL is sufficient to generate and recognise the patterns. On the other hand, there are other instances, such as the phrasal verbs and word reduplications (as shown above), which require a combination of *Kleene Closure*, the *Don't-Care* symbol and *Back-Referencing* (a special device by which we can encode a class of word-reduplication as the pattern `(%rnr)` where `r` is the reduplicated word) [4].

An interesting theoretical question to ask here is — can we classify lexical patterns based on the degree with which they exercise the aspects of RL operators? This theoretical question has a further implication to the implementation of such pattern recognition machines.

Principle 1: Each class of lexical patterns should be handled by different classes of machinery to take advantage of the computing efficiency.

In this paper, we discuss the concepts and component techniques of a Regular Expression Segmentor (REXSEG) which adheres to Principle 1. A Lexical Pattern Command Language (LPCL) was designed and implemented as a grammar to assist computational linguists in matching and processing significant lexical patterns in texts. Finally we detail comparisons with other lexical pattern matchers in use today and the future development of REXSEG.

## 2 Regular Expression Segmentor

### 2.1 Background

REXSEG is an implementation of a lexical pattern matcher — a compiler which lexically analyses and parses a set of phrasal patterns defined by the LPCL. Intermediate code is generated for matching phrasal patterns in normal texts. REXSEG allows application development programmers and computational linguists to develop sophisticated lexical analysers and parsers based on extended RL processing techniques.

### 2.2 Regular Language

Regular Language also called Regular Expression (RE) and Regular Set. It can be defined abstractly as in the following:

1. The empty set is a RE.
2. *Epsilon* is a RE and denotes the set *{Epsilon}*.
3. For each *a* in the alphabet set **A**, *a* is a RE and denotes the set *{a}*.
4. If *r* and *s* are REs denoting the languages **R** and **S** respectively, then *(r + s)*, *(r.s)* and *(r*)* are REs that denote the sets **R** ∪ **S**, **RS**, and **R*** respectively.

Where,
+ is called the *Union* (or disjunction) operator
. is called the *Concatenation* operator, and
* is called the *Kleene Closure* (or closure) operator

The LPCL is modelled after RL with extension to incorporate feature structure matching for each atom specified in a pattern. For example, the pattern condition `((a|b)c(d|e))` means `(a+b).c.(d+e)` in formal terminology.

### 2.3 Finite State Automata

REXSEG follows the design Principle 1 in classifying the degree of morphological richness of the patterns — from the lowest class (such as dead idioms) to the highest (such as person names in South East Asia). These exercise all aspects of RL operators and the *Don't-Care* symbol.

The mechanism used to recognise RL is the Finite State Automata (FSA) or equivalently the Finite State Machine (FSM). REXSEG assumes the principle of RL, hence the phrasal patterns specified in REXSEG follows the category of RL which is compiled into an FSA as a back-bone for the pattern-matching process. For the lower class, an acyclic-FSM pattern matching mechanism is applied [1] while an FSM-based pattern matching mechanism is applied in the higher class.

### 2.4 Lexical Pattern Command Language

LPCL is a grammar for writing patterns and corresponding actions for the pattern that has been matched, with facilities for macro and feature definitions. Several patterns and associated actions can be specified for the lexical matching process. A lexical pattern can comprise of,

- pattern or macro name

- pattern condition - specified based on an extended regular expression grammar
- constrained and unconstrained variables - which specify feature structures for words
- pattern actions - program codes for associated patterns

A pattern definition is a list of atoms or variables specifying a pattern for matching. It uses an extended regular expression grammar, achieved by adding syntactic sugaring to expressions. A Constrained Variable is matched if its specified features are satisfiable (the variable's features match the token's features based on the specified binary operation). An Unconstrained Variable and its *Kleene Closure* do not have any associated features and hence is similar to the *Don't Care* symbol. An example of a pattern definition is as follows,

```
%feature word char*
%feature root char*
%feature pos  char*
(Macro
 (a(b|c))
 (a.root="please" b.word="give" c.word="take")
)
(Pattern
 ({Macro}(1*)ab+)
 (a.word="up" b.pos="NOUN")
 -->
 {  /* Pattern actions (ANSI C codes) */  }
)
```

where `Macro` and `Pattern` are the names, `(a(b|c))` and `({Macro}(1*)ab+)` are the conditions, `(a.root="please" b.word="give" c.word="take")` and `(a.word="up" b.pos="NOUN")` are the variable constraints for macro and pattern definitions respectively. Note that `word` (surface word), `root` (root form) and `pos` (part-of-speech) are defined features associated with constrained variables `a`, `b` or `c`. `(1*)` is an unconstrained variable with *Kleene Closure* — matching zero or more tokens. Pattern actions manipulate the matched portions of text and are coded in ANSI C. `Pattern` can thus recognise morphologically rich tokens such as "Please give up Mary", "please take it up Mary", "please give it all up John", "Please take this up John Doe" and many more phrasal verbs with a common noun.

The following is a formal description of the LPCL grammar based on an extended BNF with regular expression syntax.

```
/* LPCL Top-Level Program Structure */
PROGRAM := {DECLARATION}1+ "%%"  PROG_BODY  "%%"  PROG_TAIL
PROG_BODY    := {FEATURE}1+  {MACRO}0+  {PATTERN}1+
DECLARATION := /* ANSI C file includes and variable declarations */
PROG_TAIL    := /* ANSI C main() and other associated functions */

/* Feature Structure */
FEATURE := "%feature"  NAME  FEATURE_TYPE
FEATURE_TYPE := "uchar"|"ushort"|"ulong"|"int"|   /* long form  */
                "float"|"double"|"char\*"|"NULL"|
                "uc"|"us"|"ul"|"i"|"f"|"d"|"c"    /* short form */

/* Macro and Pattern Structure */
MACRO := "("  NAME  CONDITION  CONSTRAINT  ")"
PATTERN := "(" NAME  CONDITION  CONSTRAINT "-->" PAT_ACTION ")"
PAT_ACTION     := "{" /* ANSI C codes */ "}"

/* Common Structures */
CONDITION  := EXPR EXPR |                  /* concatenation      */
              EXPR "|" EXPR |              /* disjunction        */
              EXPR "<" LOW "," HIGH ">" |  /* match within range */
              EXPR "*"                   | /* match 0 or more    */
```

```
                EXPR "+"                    | /* match 1 or more     */
                "[" EXPR "]"                  /* optional match      */
EXPR        := CS_VAR | UCS_VAR   |           /* pattern variables   */
                "{" MAC_NAME "}"  |           /* macro in pattern    */
                "(" CONDITION ")"
CS_VAR      := [A-Za-z]             /* Constrained variables        */
UCS_VAR     := [0-9]                /* Unconstrained variables      */
LOW         := [0-9]+
HIGH        := [0-9]+               /* Note: HIGH must be >= LOW    */

CONSTRAINT  := "("  (EXPR_ASG)0+  ")"
EXPR_ASG    := CS_VAR  "."  FEATURE_NAME  OP  VALUE
VALUE       := "\""  [.]+  "\"" | /* Quoted character string */
               [0-9]+           | /* integers                    */
               "<?>"              /* For CS_VAR - match anything */
OP          := "=" | "!=" | ">" | ">=" | "<" | "<="
NAME        := [A-Za-z\_][A-Za-z0-9\-\_]*
```

## 3 Discussion and Comparisons

There are several programming languages and tools available today to facilitate lexical pattern matching. Some examples of these are `lex` [6] and SNOBOL4 [3] of Bell Laboratories, `perl` by Larry Wall and Ronald L. Schwartz [9], and FDF-3 by Paracel [8]. Of particular interest, is the widely used and freely available version of `lex` called `flex`, a fast lexical analyser, developed by the GNU project. The functional concepts of REXSEG were initially based on `lex` and were further enhanced. The following details the comparison between lex and REXSEG, but more importantly significant REXSEG features.

- `lex` doesn't optimise pattern matching efficiency according to the morphological richness of lexical patterns. Hence it frequently terminates due to the overflow of states during the FSM construction process. With the notion of morphological richness, a specialised version of an acyclic-FSM is used to handle *Concatenation* only patterns.
- `lex` is known to handle only a single stream of input tokens (a single tape), in particular on a character-by-character basis. This restricts the lexical analyser from analysing linguistic features associated with a token (word). The examination of additional linguistic features along with the corresponding matched token, with potentially as many as required, aids the reliability of the lexical analysis.
- `lex` employs a maximum-match heuristic to resolve word segmentation ambiguity. REXSEG, however, enumerates all possible candidates for a matched pattern, allowing one to have the freedom of implementing resolution strategies for disambiguation.
- `lex` doesn't provide a convenient way of accessing sub-patterns of a complex pattern, however REXSEG uses variable and macro bindings in pattern definitions and hence allows sub-patterns access by referencing the bindings of the variables or macros.
- `lex` does not provide back-referencing, which is crucial for handling word-reduplication in Chinese, Malay and other Asian languages.

SNOBOL4, an acronym for String Oriented Symbolic Language, was initially designed by Bell Laboratories in the 1960s as a programming language for a wide range of non-numeric applications - in particular string matching processes. Features of this language include no type declaration, automatic type checking and conversion, dynamic memory allocation and garbage collection, pattern assignment, program modification during execution and associative tables. SNOBOL4 was designed for linguists and softcore programmers. The high-level language structure doesn't incorporate compiler details like type-checking and memory management to confuse the "programmer". However the programming language is unstructured, much like building prototypes without detailed input and output specifications.

The perl programming language facilitates manipulation of text, files and processes. It provides a more concise and readable way to handle jobs accomplished by less intuitive languages like C or UNIX shells. This rich language has extensive capabilities, borrowing from UNIX shells, the C language and is a superset of sed and awk. The pattern matching processes make use of an extended regular expression grammar (currently Version 8 regexp routines). perl claims that it is intuitive for one to program in such a language however, due to the richness and scope of this programming language, it would be a daunting task for computational linguists to fully utilise its functionalities for Natural Language processing.

FDF 3, an acronym for Fast Data Finder, developed by Paracel is claimed to be the world's fastest pattern-matching accelerator for real-time information analysis and selective distribution. It comprises of hardware — a pipeline VLSI chip design with patented pattern matching algorithm, and software — a query language called Pattern Specification Language (PSL). The PSL includes manipulation of lexical items and patterns (no recursive expressions), pattern ranges, text "windows", Boolean operators, error tolerance, sets, lists, counting and macro definitions. FDF 3 was built solely for fast and efficient pattern matching and handling for any domain that requires such functionality.

As far as pattern matching proper is of concern, none has tried to cover a fuller set of operators and aspects of RL than REXSEG. Feature structure matching is only supported by REXSEG; the other four matches only the surface form of the input tokens. REXSEG can be a standalone pattern matcher or be incorporated into larger NL processing aplications.

## 4 In The Future

These are the few items in the pipeline for further development of the REXSEG system.

- Incorporate the UNICODE technology developed in ISS [7] into REXSEG to provide a full multilingual pattern matching capabilities
- Kernel tuning to make REXSEG more efficient in performance and use of resources
- User-friendly LPCL design with comprehensive syntax checking and higher-level language for specifying pattern actions instead of ANSI C codes

## References

[1] Aho, A.V. & Corasick, M.J.(1975). Efficient string matching: an aid to bibliographic search. Comm. ACM 18(6) 333-340
[2] Chomsky, Noam (1963). Syntactic Structures. The Hague, Holland: Mouton.
[3] Griswold, Ralph & Griswold, Madge T.(1973). A SNOBOL4 Primer. Prentice-Hall, Inc.
[4] Jan Van Leeuwen (1990). Algorithms and Complexity. MIT Press.
[5] Johnson, Eric (1990). ProIcon and MaxSPITBOL - The Languages of Choice for Non-Numeric Computing. Dakota State University. Bits and Bytes Review, 1-11.
[6] Levine, John R., Mason, Tony & Brown, Doug (1992). lex and yacc. O'Reilly & Associates, Inc.
[7] MASS (1995) Multilingual Application Support Service (MASS) version 2.4 Programmer's Reference Guide.
[8] Paracel, Inc (1992). FDF 3 - Technical Description and PSL Reference Manual.
[9] Wall, Larry & Schwartz, Randal L. (1991). Programming perl. O'Reilly & Associates, Inc.