Optimization of Regular Expression Evaluation within the Manatee Corpus Management System

Miloš Jakubíček and Pavel Rychlý

Natural Language Processing Centre
Faculty of Informatics, Masaryk University
Botanická 68a, 602 00 Brno, Czech Republic
{jak,pary}@fi.muni.cz

Lexical Computing Ltd.
Brighton, United Kingdom
{milos.jakubicek,pavel.rychly}@sketchengine.co.uk

Abstract.
This paper is concerned with searching large text corpora – electronic collections of texts. Often these are subject to queries specified by means of regular expressions. Such queries go beyond a simple keyword search that can be quickly evaluated using an inverted index, usually they are rather processed by third-party regular expression libraries and take significantly more time to evaluate. In this paper we present an index-based approach for optimization of regular expression evaluation that we call *n-gram prefetching*. It is based on the assumption that most regular expression queries on text corpora contain at least some fixed string portions representing clues that can be used for developing heuristics that would prune the number of potentially matching strings. The presented work has been designed and implemented within the Manatee corpus management system. We show that the proposed approach can significantly speed up regular expression processing by providing evaluation on a test set of queries executed on a number of billion-word text corpora.

Keywords: text corpus, regular expression, Manatee

1 Introduction

Text corpora represent primary data resource for testing hypotheses, providing evidence and building large scale statistical models used in various natural language processing applications such as part-of-speech tagging, parsing or machine translation.

Special database management systems (in this case corpus management systems) devised for indexing and querying large text corpora have been developed to satisfy user needs in terms of complexity of queries and related response time, such as [1,2]. These corpus management systems usually leverage the idea of inverted index (inverted text) [3] to provide fast access to all occurrences of a given word (or another attribute like lemma or tag depending on the type of annotation) in the corpus.

Aleš Horák, Pavel Rychlý (Eds.): Proceedings of Recent Advances in Slavonic Natural Language Processing, RASLAN 2014, pp. 37–48, 2014.

In many cases users formulate the queries not in the form of fixed string expressions but as regular expressions. Clearly this represents a serious challenge for the corpus management systems as the first step necessary to answer such queries lies in evaluating the regular expression against the relevant lexicon used in the corpus so as to be able to retrieve matching strings from the indices. Usually, a third-party regular expression library is used for the actual matching, often providing expressive power going way beyond what regular expression offer as a classical computational model.

In this paper we present an optimization of regular expression evaluation that we call *n-gram prefetching*. The approach has been implemented within the Manatee corpus management system [4] and exploits the PCRE regular expression library [5], however we claim that it is suitable for any inverted-index-based corpus management system and it is in no way dependent on any particular regular expression library.

The structure of this papers is as follows: we first provide a brief overview of the Manatee corpus management system and its overall indexing machinery, then we present the optimization approach in detail and finally we provide and evaluation on a number of billion word corpora showing a up to 100-times speedup.

2 Manatee

Manatee is a state-of-the-art corpus management system providing facilities for efficient indexing (compiling) and searching billion-word-sized corpora [6]. Querying corpora indexed by Manatee is done using the Corpus Query Language (CQL, [7]). From a formal perspective a corpus in manatee consists of *text data* (called tokens or positions, each of which may be associated with a number of attributes such as word, tag or lemma, further referred to as positional attributes) and *text metadata* (called structures, each of which is denoting a span in the corpus such as a document, paragraph or sentence, and may be associated with arbitrary number of structure attributes, denoting e.g. the author of a document, date of creation etc.).

Every positional or structural attribute possesses following basic index structures:

- **attribute lexicon** providing efficient string↔ID mapping. Each unique attribute string value is assigned a unique numeric ID which is further used in all indices and for processing CQL queries,
- **attribute inverted (reversed) index** providing sequential access to a sorted list of occurrences (corpus positions) of a given attribute ID,
- **attribute text** storing the actual text of this corpus attribute, i.e. a sequence of attribute IDs in the order of occurrence in the corpus.

On a very abstract level evaluating a CQL query consists of mapping attribute strings given in the query to their IDs (using the lexicon index), retrieving the relevant positions from the inverted index, combining them

according to the given CQL operators and displaying the results (e.g. in the form of a concordance). In the simplest case, considering a query looking for all occurrences of a single word in corpus like `[word="someword"]`, one yields the ID n of `someword` from the lexicon and then retrieves the sorted list of positions for n from the inverted index. Further CQL operations always rely on processing sorted streams (of corpus positions or attribute IDs).

In the case attribute constraints are given as regular expressions, the straightforward string-to-number mapping using the lexicon is not possible. First one needs to find out which attribute values are matching the given regular expression (by scanning the whole lexicon), then map each of the values to the respective ID and merge all the related position streams retrieved from the inverted index. Since all the streams and results must be sorted by design, before the matching has finished no results are available to the user. This exhibits a serious issue for large (i.e. billion-word-sized corpora) corpora where the lexicon size is often reaching tens of millions of values and hence the time taken to evaluate the regular expressions across the whole lexicon represents a significant slow down of query evaluation, and especially of retrieving first n results.

Two basic optimization have already been in place to tackle this problem:

- **any string optimization**: a regular expression matching `^(\.\*)+$` was omitted as it obviously must match the whole lexicon,
- **prefix optimization**: since the lexicon provides an index of attributes ID sorted alphabetically by the corresponding string values (representing one of the possible implementations of string↔ID mapping based on a simple binary search, see [8] for comparison), all regular expressions containing a fixed-string prefix like `re.*` or `mis.*ing` make it possible to shrink the set of possible matching strings by selecting the range of IDs with the given prefix (here `re` and `mis`, respectively).

3 n-gram prefetching

In this paper we describe a new optimization approach that we call *n-gram prefetching*. It is based on the assumption that most user queries exploiting regular expressions still contain some fixed-string portions (because they are linguistically motivated). While queries like `^.{0,3}$` are of course possible, they are very rare. To verify this hypothesis we have inspected a set of 128,406 queries which the users of Sketch Engine (a web service exploiting Manatee, see [9]) have issued to that system over the period from June to September 2014. Only 12 of them did not contain any fixed-string portions, moreover 6 of these 12 were `^.*$` and got optimized as well.

The idea of n-gram prefetching consists (on compile time) in indexing all character uni-, bi- and trigrams of every string attribute value and (on run time) in extracting such n-grams from the regular expressions and using them to constrain the number of possibly matching strings from whole lexicon to a much smaller set of IDs.

3.1 n-gram indexing

For indexing of the character n-grams we leverage the idea of *dynamic attributes* in Manatee. A positional or structural attribute in Manatee can be a so called *dynamic attribute* in which case such an attribute is automatically derived from another existing (regular or dynamic) attribute. Each dynamic attribute is assigned a dynamic function which takes the source attribute string as input (and optionally other parameters as well) and returns the new (dynamic) attribute value. Manatee contains a predefined set of dynamic functions which mostly focus on simple string manipulations [1] (such as getting a prefix or suffix of a string or its lowercase variant) and users can supply their own dynamic functions as well (in the form of Linux plugins – dynamically linked C/C++ libraries). The main benefits of a dynamic attribute are:

- **space savings in source data:** no need to make it part of the input vertical text
- **space savings in indexing:** a dynamic attribute has only the lexicon and inverted index, but no text index. The inverted index stores for each dynamic attribute ID a sorted list of source attribute IDs which map to this dynamic attribute ID (instead of storing corpus positions).
- **very limited runtime overhead:** depending on the type of operations, the overhead (slowdown) of using a dynamic attribute instead of a regular one is very small. Optionally an index providing mappings of source attribute ID to dynamic attribute ID is compiled as well, in which case the dynamic functions do not need to be executed at runtime at all (except where it is necessary to convert input user query, e.g. in case of lowercasing).

Each lexicon item (string) is processed generating all occurring uni-, bi- and trigrams as shown in Figure 1 and storing the n-gram values as a dynamic attribute of the source attribute.

```
classical  ->  c|l|a|s|s|i|c|a|l
               cl|la|as|ss|si|ic|ca|al
               cla|las|ass|ssi|sic|ica|cal
```

Fig. 1: Uni-, bi- and trigram string generation.

The resulting dynamic attribute contains a lexicon with all the n-grams found in the source attribute and provides fast access to all source attribute IDs containing a given n-gram. We prepend the caret sign (`^`) and append the dollar sign (`$`) to each string so that we generate specific n-grams occurring at beginnings and ends of words.

[1] See `https://www.sketchengine.co.uk/documentation/wiki/SkE/Config/DynamicAttributes`.

Table 1: Comparison of original and n-gram lexicon sizes

corpus	language	size $\times 10^9$	attribute	lexicon size	n-gram lexicon size
czTenTen12	Czech	5.126	word	18,978,703	522,745
			lemma	14,151,454	506,580
			tag	12,061	1,702
enTenTen12	English	12.968	word	27,894,538	1,880,911
			lemma	26,426,200	1,880,808
			tag	60	260
enClueWeb09		82.581	word	115,820,931	2,350,697
			lemma	110,606,268	2,296,072
			tag	60	260
jpTenTen11	Japanese	10.322	word	13,844,200	6,353,186
			lemma	13,303,479	3,766,160
			tag	53	297

3.2 n-gram matching

On runtime we parse the given regular expression and extract all occurring fixed-string uni-, bi- and trigrams (preferring longer n-gram where available and combining strings longer than 3 characters into a set of trigrams by the logical `AND` operator in CQL). Since regular expression as such can be quite complicated we exploit the ANTLR3 parser generator [10] for processing the regular expressions. We have chosen ANTLR3 because it has already been used within Manatee (for CQL and corpus configuration files parsing) and it has very powerful grammar writing formalism. The ANTLR3 lexer and parser grammar is provided in Annex 1. The output of the ANTLR3 lexing and parsing is an abstract syntax tree (AST) that is further subject to parsing by ANTLR3 using a so called tree walker which directly executes programming code according to the parsed AST.

The regular expression parsing grammar is based on the following principles:

- it recognizes separately regular characters and metacharacters (except for ^ and $ which are intentionally indexed as part of the n-grams as explained above), because metacharacters must not be included into the n-grams search and hence represent a separator between n-grams,
- it recognizes character classes (enclosed in [and] brackets) which must be kept as a single token as they represent a single character position,
- it recognizes escaped sequences so that they can be handled correctly (possibly de-escaped),
- it recognizes repetitions which must be either entirely omitted (in case zero number of repetitions is allowed and hence the respective string portion

is entirely optional) or if at least one occurrence is obligatory, the operand character is duplicated and forms a suffix of previous n-grams and prefix of next n-gram. E.g. `abc+de` gets expanded into two n-grams `abc` and `cde` since every matching string must contain these.

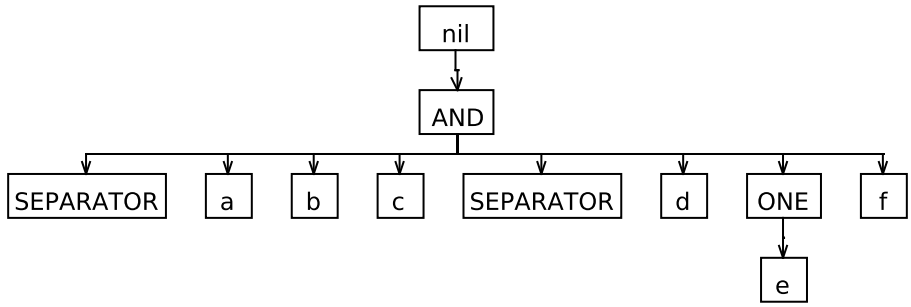

Fig. 2: Sample abstract syntax tree for the input query `.abc.*de+f`. Both `.` and `.*` substrings become a separator, while `e+` substring is recognized so as to search for `de` and `ef`.

Two sample AST's are provided in Figures 2 and 3. The tree walking parser looks up the found n-grams (in this sample `abc`, `de` and `ef` and combines the related source attribute IDs using the logical `AND` operator into a single stream of IDs that represent possibly matchings strings. Only these IDs are then subject to evaluation of the regular expression instead of the full lexicon. An overview of the whole dataflow is presented in Figure 4.

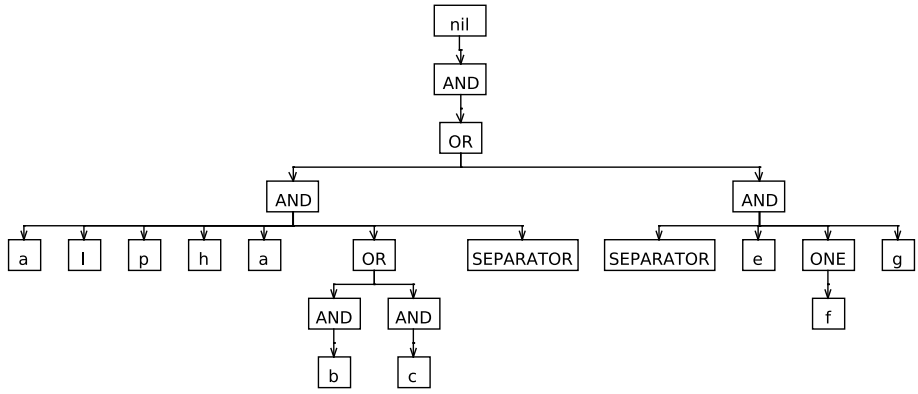

Fig. 3: Sample abstract syntax tree for the input query `(alpha(b|c).*|d*ef+g)`

The lexicon lookup for particular n-grams is either a direct mapping of an n-gram to ID using the dynamic attribute's lexicon, or in case of n-grams containing regular expression character class again an evaluation of a regular expression – however on a much smaller lexicon. In Table 1 we provide a comparison of original and n-gram lexicon sizes of various corpus and attribute combinations showing that the n-gram lexicon is usually by orders of magnitude smaller. It is obvious that for attributes with very small lexicons (such as tags based on atomic tagsets), the optimization is not worth doing and may even slow down the processing (as in the case of English and Japanese tags), therefore in the current implementation the n-gram indices are being compiled only for attributes with lexicons exceeding 10,000 items.

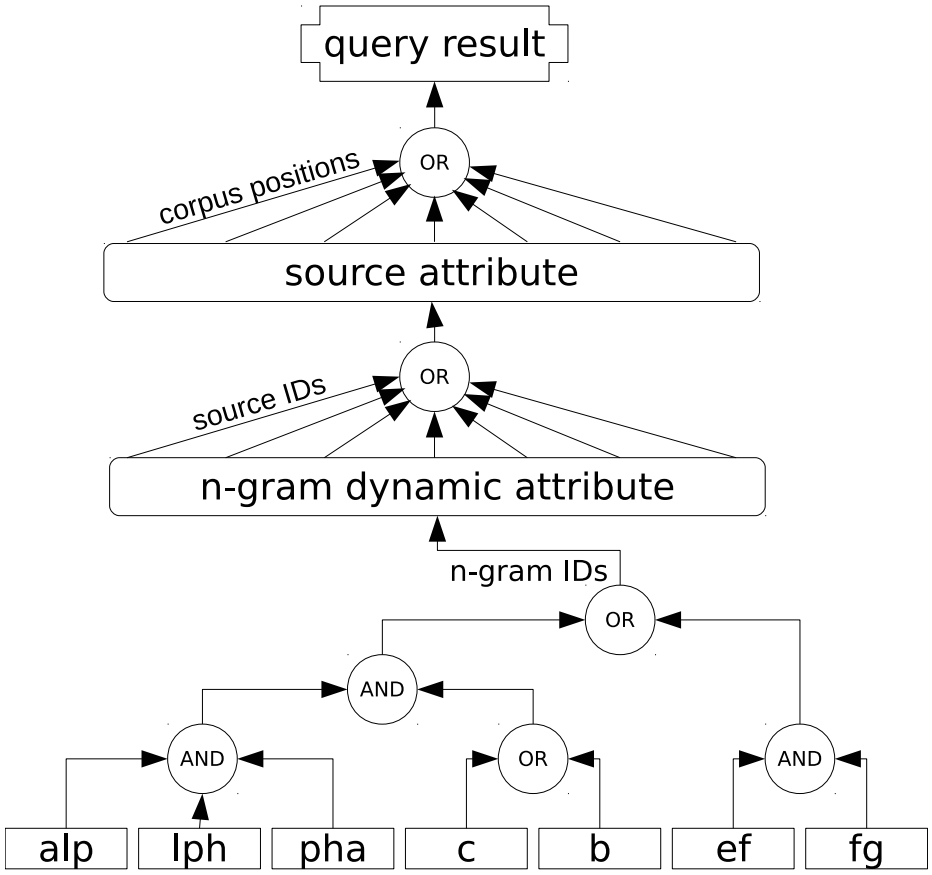

Fig. 4: Overview of the evaluation workflow using the n-gram prefetching optimization for input query `(alpha(b|c).*|d*ef+g)`

4 Evaluation

The optimization has been evaluated on a number of regular expressions executed against various corpora under the following conditions:

- a single evaluation thread running on an Intel(R) Xeon(R) CPU E5506 @2.13GHz,
- hot cache – the best time of three consecutive runs was counted,
- we have measured the time to retrieve first 20 hits – this includes the regular expression evaluation plus a number of other operations on the resulting positions stream but it is more representative from the user perspective (the processing within Sketch Engine is asynchronous and as soon as first 20 hits are available they are displayed to users),
- we provide the number of regular expression evaluations (calls to the PCRE matching function) with and without the optimization.

It follows from the evaluation that the speedup ranges from rather negligible 1.12 to enormous 100-times and the speedup ratio depends on a number of circumstances:

- obviously the larger the lexicon size of the source attribute, the more speedup can be achieved, and the evaluation shows that the additional indexing pays off only in cases where the lexicon size exceeds about 10,000 items
- for very small lexicons (e.g. in the case of the tag attribute in English corpora following the Penn Treebank tagset with only 60 atomic tags), the n-gram prefetching is not very beneficial as it enlarges the lexicon size,
- the n-gram prefetching is beneficial even in cases where the prefix optimization has already been previously in operation which is important since these two optimizations cannot be combined,[2]
- not surprisingly the n-gram prefetching is most beneficial for regular expressions containing rare n-grams (e.g. `strč` in Czech) but even for frequent n-grams (like `ten` in English) the speedup is usually around 20,
- even where the optimization itself involves regular expression evaluation (character class matching as in `[sz]p.*`), the speedup is significant and present even in comparison with prefix optimization (as in `pr[oe].*`).

5 Technical Notes

For creating the n-gram optimization index, there is a new tool included in Manatee called `mkregexattr` with a straightforward usage:

```
mkregexattr <CORPUS> <ATTRIBUTE>
```

[2] The prefix optimization results – by its nature – in a list of IDs sorted alphabetically, not numerically, and hence cannot be as such subject to any AND/OR stream operations.

Table 2: Evaluation of n-gram prefetching optimization. #RE denotes the number of regular expression matching function executions without and with n-gram prefetching, time is the evaluation time to acquire first 20 hits in seconds, S denotes the achieved speedup.

corpus	query	#RE w/o	#RE w/	time w/o	time w/	S
czTenTen12	`[word=".*ější"]`	18,978,703	41,426	10.030	0.341	29.4
	`[lemma=".*strč.*"]`	14,151,454	888	6.601	0.066	100.0
	`[tag="k1.*c4.*"]`	1,357	251	0.058	0.049	1.2
	`[word="[sz]p.*"]`	18,978,703	115,347	10.023	0.698	14.4
enTenTen12	`[word=".*ing"]`	27,894,538	913,004	21.931	2.768	7.9
	`[lemma=".*ten.*"]`	26,426,200	195,758	22.163	1.218	18.2
	`[word="pre.*ed"]`	80,054	7,553	0.294	0.178	1.7
	`[word="pr[oe].*"]`	251,924	198,297	1.329	0.920	1.4
	`[word=".*[dt]"]`	27,894,538	3,466,379	41.100	8.538	4.8
	`[tag="N.*"]`	60	5	0.056	0.048	1.2
jpTenTen11	`[word=".*ち.*"]`	13,844,200	30,160	8.182	0.364	22.4
	`[lemma=".*ア.*ス"]`	13,303,479	69,228	8.388	0.450	18.6
	`[word="ンテ.*"]`	17,078	17,077	0.199	0.178	1.12

The tool is part of Manatee version 2.111 and since this version, it is called automatically by `encodevert` at the end of corpus compilation for each attribute whose lexicon size exceeds 10,000 items.

6 Conclusions and Future Work

In this paper we have presented n-gram prefetching – an optimization approach for regular expression evaluation suitable for any corpus management system based on inverted indices and independent of any third-party regular expression library. We have shown that this optimization can significantly speedup user queries consisting of regular expressions. The idea has been practically implemented within the Manatee corpus management system used within the Sketch Engine corpus system and is part of the GPL-licensed part of Manatee available also within the open source NoSketch Engine suite[3].

In the future there might be a number of further optimizations that could be explored, starting with extending the character class recognition support to escape sequences like `\w`, `\d` etc., or dividing the n-gram lexicon into separate ones for uni-, bi-, and trigrams. A different kind of optimization may also lie in trying different regular expression library than PCRE or enabling PCRE just-in-time (JIT) features that are currently not in use – however such a contribution

[3] `http://nlp.fi.muni.cz/trac/noske`

is hard to evaluate as it very much depends on particular types of regular expressions and with regard to them the speedup might be fairly unstable.

Acknowledgements This work has been partly supported by the Ministry of Education of CR within the LINDAT-Clarin project LM2010013.

References

1. Rychlý, P.: Korpusové manažery a jejich efektivní implementace. PhD thesis, Fakulta informatiky, Masarykova univerzita, Brno (2000)
2. Evert, S., Hardie, A.: Twenty-first century corpus workbench: Updating a query architecture for the new millennium. (2011)
3. Knuth, D.E.: Retrieval on secondary keys. The art of computer programming: Sorting and Searching **3** (1997) 550–567
4. Rychlý, P.: Manatee/Bonito - A Modular Corpus Manager. In: Proceedings of Recent Advances in Slavonic Natural Language Processing 2007, Brno, Masaryk University (2007)
5. Hazel, P.: PCRE: Perl compatible regular expressions (2005)
6. Pomikálek, J., Rychlý, P., Jakubíček, M.: Building a 70 billion word corpus of English from ClueWeb. In: Proceedings of the Eight International Conference on Language Resources and Evaluation (LREC'12). (2012) 502–506
7. Jakubíček, M., Rychlý, P., Kilgarriff, A., McCarthy, D.: Fast syntactic searching in very large corpora for many languages. In: PACLIC 24 Proceedings of the 24th Pacific Asia Conference on Language, Information and Computation, Tokyo (2010) 741–747
8. Jakubíček, M., Šmerk, P., Rychlý, P.: Fast construction of a word-number index for large data. In A. Horák, P.R., ed.: RASLAN 2013 Recent Advances in Slavonic Natural Language Processing, Brno, Tribun EU (2013) 63–67
9. Kilgarriff, A., Baisa, V., Bušta, J., Jakubíček, M., Kovář, V., Michelfeit, J., Rychlý, P., Suchomel, V.: The Sketch Engine: ten years on. Lexicography **1** (2014)
10. Parr, T.: The definitive ANTLR reference: building domain-specific languages. Pragmatic Bookshelf (2007)

Annex 1: ANTLR3 lexer and parser grammar for regular expressions

```
LPAREN:      '(';
RPAREN:      ')';
LBRACKET:    '[';
RBRACKET:    ']';
LBRACE:      '{';
RBRACE:      '}';
BINOR:       '|';
STAR:        '*';
PLUS:        '+';
QUEST:       '?';
DOT:         '.';
ZEROANDMORE: '{0,}' | '{0,' ('0'..'9')+ '}';
ONEANDMORE:  '{1,}' | '{1,' ('0'..'9')+ '}';
ESC:         '\\' (STAR | PLUS | QUEST | LBRACE | RBRACE | LBRACKET | RBRACKET
                   | LPAREN | RPAREN | DOT | BINOR | '\\' | '^' | '$'
                   );
BACKREF:     '\\' ('0'..'9')+;
SPECIAL:     '\\' .;
NOMETA:      ~(STAR | PLUS | QUEST | LBRACE | RBRACE | LBRACKET | RBRACKET
               | LPAREN | RPAREN | DOT | BINOR | '\uFFFF'
               );
ENUM: LBRACKET CHARCLASS+ RBRACKET;
fragment CHARCLASS: ( (NOMETA|DOT) '-' (NOMETA|DOT)
           | ALNUM | ALPHA | BLANK | CNTRL | DIGIT | GRAPH | LOWER
           | PRINT | PUNCT | SPACE | UPPER | XDIGIT
           | (NOMETA|DOT)
           );


regex
    : regalt (BINOR^ regalt)* EOF!
    ;

regalt
    : regpart+ -> ^(AND regpart+)
    ;

regpart
    : regterm
      ( repet_one -> ^(ONE regterm)
      | repet_zero -> SEPARATOR
      | -> regterm
      )
    ;

regterm
    :
```

```
    (
        LPAREN regalt (
                        (BINOR regalt)+ -> ^(BINOR regalt regalt)
                        | -> regalt
                      ) RPAREN
    |   ENUM -> ENUM
    |   re_str -> re_str
    )
    ;

re_str
    : (DOT|BACKREF|SPECIAL) -> SEPARATOR
    | NOMETA -> NOMETA
    | ESC -> ESC
    ;

repet_one
    : PLUS -> PLUS
    | ONEANDMORE -> ONEANDMORE
    ;

repet_zero
    : QUEST -> QUEST
    | STAR -> STAR
    | ZEROANDMORE -> ZEROANDMORE
    ;
```