Converting regexes to Parsing Expression Grammars

Marcelo Oikawa[1], **Roberto Ierusalimschy**[1], **Ana Lúcia de Moura**[1]

[1]Departamento de Informática, PUC-Rio,
R. Marquês de São Vicente 225, Gávea,
Rio de Janeiro, 22453-900.

{moikawa, roberto}@inf.puc-rio.br, analuciadm@gmail.com

Abstract. *Most scripting languages offer pattern-matching libraries based on extensions of regular expressions. These extensions are mainly composed of ad-hoc constructions that aim at specific problems, resulting in complex implementations without a formal basis. Parsing Expression Grammars (PEGs) provide a well-founded alternative for pattern recognition, with more expressive power than pure regular expressions. In this work we present a new algorithm to convert regular expressions into PEGs and discuss the conversion of some common regular-expression extensions, such as captures and independent subpatterns. Besides its practical applications, this conversion algorithm sheds some light on the semantics of those extensions.*

1. Introduction

Most scripting languages offer pattern-matching implementations based on extensions of regular expressions. These extensions, called *regexes*, aim to solve limitations in pure regular expressions when used for pattern-matching. Among the most common constructions found in these implementations we have anchors, independent subpatterns, captures, and lookaheads.

Regexes are useful to describe several patterns in practice but they bring their own set of problems. Because they incorporate an aggregate of features with no formal basis, it is difficult to infer the semantics of some combinations and also to predict the performance of any particular pattern.

Parsing Expression Grammars (PEGs) [Ford 2004] are a recognition-based foundation for describing syntax, and provide an alternative for context-free grammars. Recently, one of the authors [Ierusalimschy 2009] has proposed that PEGs also provide an interesting alternative for pattern recognition, with more expressive power than pure regular expressions.

The absense of a formal basis for regex constructions and the expressive power of PEGs motivated us to investigate a translation from regexes to PEGs. Our insight for this investigation was that PEG operators, besides having similarities with pure regular expressions, also have similarities with regex constructions, such as lookaheads, possessive quantifiers, and independent expressions. One result from this translation is a better understanding of the semantics of regexes, supported by the formal model of PEGs. Another result is the possibility to execute regexes using a PEG engine.

This paper presents a new algorithm to convert regular expressions into PEGs, called *continuation-based conversion*, and also discusses the conversion of some common

regular-expression extensions. It is organized as follows: section 2 discusses some typical regex constructions. Section 3 reviews PEG concepts. Section 4 presents our algorithm for converting regular expressions into PEGs and discusses the conversion of some regex extensions such as captures and independent expressions. Finally, section 5 draws some conclusions.

2. Regexes

Regular expressions are a powerful tool for describing regular languages, but are too limited for creating some patterns in practice. Most pattern-matching implementations are then based on combinations of regular expression operators, shown in table 1, with additional constructions that address specific pattern-matching needs.

e	L(e)	Operation
ϕ	ϕ	empty language
ε	{""}	empty string
a	{"a"}	alphabet symbol $a \in \Sigma$
$e_1 \mid e_2$	$L(e_1) \cup L(e_2)$	alternation
$e_1\, e_2$	$L(e_1)\, L(e_2)$	concatenation
e^*	$L(e)^*$	repetition

Table 1. Regular expression constructors

Perl's *regex* [Truskett] popularized the use of pattern-matching based on regular expression extensions, and introduced several constructions that are the basis for most current regex libraries, such as Python's [Lutz 2006], Ruby's [Thomas et al. 2009], Java's [Habibi 2004], and the PCRE library [Hazel 2009]. These implementations are based on backtracking, which allows them to determine exactly how a matching was found, and thus support captures and backreferences.

The rest of this section presents some common regex extensions provided by Perl's regex. As we will see, some of these constructions do not have a clear semantics and when we combine them unexpected behaviors can occur.

2.1. Independent expressions

An independent expression (sometimes called *atomic grouping* [Friedl 2006]) is a construction that allows a subexpression to match independently of its enclosing expression. If an independent expression matches, it will not be subject to backtracking, even if its enclosing expression matching fails. Independent expressions are denoted by `(?>p)` where `p` is a pattern.

Making subexpressions match independently has some interesting practical consequences, notably for pattern optimizations. Because an independent expression is not subject to backtracking, it is possible to avoid trying useless matches when no overall match is found. As an example, consider the expression `\b(?>\d*)\b` where `\b` matches word boundaries and `\d` matches a digit. When we match this expression against the string `"123abc 456"`, the first `\b` matches at the start of subject and the subpattern

`(?>\d*)` matches `"123"`, but the second `\b` fails to match between `"3"` and `"a"`. Because the subpattern is an independent expression, there will be no further attempts to match it with shorter sequences, which would not lead to a successful overall match.

Note that the use of independent subexpressions does not result only in a more effective matching process. Whether a subexpression is an independent expression affects the semantics of its enclosing expression, providing different matches or preventing some.

2.2. Quantifiers

Quantifiers are used to express pattern repetitions. All the pattern-matching engines studied offer at least two types of quantifiers: greedy and lazy.

Greedy quantifiers express the usual pattern repetition, and always try to match the maximum possible span. Repetitions are *non-blind*: they occur as long as the rest of the pattern matches. As an example, let us consider the expression `.*10` that matches a (possibly empty) string of characters followed by the substring `"10"`. In order to permit a match for the entire subject `"May 2010"`, the repetition subexpression `.*` will match the substring `"May 20"`, leaving the rest of the subject to be matched by the subsequent pattern `10`.

Lazy, or non-greedy, quantifiers (also called *reluctant*) always try the shortest possible match that permits sucessful subsequent matches. A typical use of lazy quantifiers is to obtain the value of a tag. A naive solution would use a greedy repetition like in `<b>.*</b>`. If matched against the string `"<b>first</b><b>second</b>"` this repetition will match `"first</b><b>second"` instead of the desired result `"first"`. An adequate expression for this task can be built with a lazy quantifier, like `<b>.*?</b>`, which would match only the first tag value.

A third type of quantifier, called *possessive*, is offered by Perl's and Ruby's regex implementations. Possessive quantifiers match the longest possible string, but never backtrack, even if doing so would allow the overall match to succeed.

Because independent expressions prevent backtracking, they can express possessive quantifiers. As an example, the possessive quantifier `(ab|c)*+` can be expressed by the independent subexpression `(?>(ab|c)*)`.

2.3. Lookahead

Lookaheads are used to check whether a pattern is followed by another without consuming any input. Lookaheads are classified as *zero-width assertions*: zero-width because they do not consume any input and assertion because they check a property of the matching.

A positive lookahead is denoted by `(?=p)`, where `p` is a pattern. A negative lookeahead is denoted by `(?!p)`. The expression `p(?=q)` matches a subject if `q` matches after `p`, but `q` does not consume any portion of the input string. If a negative lookahead is used, like in `p(?!q)`, there can be no match for `q` after `p` for the overall match to succeed.

A simple use for a lookahead is an expression that matches words followed by a comma, like `\w+(?=\,)`. We can also express the final anchor (`$`), which matches the subject's end, using the negative lookahead `p(?!.)`. This expression matches only at

the end of the subject, where pattern `.` (which matches any character) does not match because no more characters are available.

2.4. Captures

Regular expressions were originally introduced for describing languages. Therefore, when we use regular expressions for pattern-matching, the basic ideia is to recognize a particular set of strings in the subject.

However, in most practical uses of pattern-matching, it is usually necessary to determine the structure of the given subject and retrieve the portions that matched specific patterns. To accomplish this task, regex implementations provide *captures*.

To be able to support captures, a pattern-matching engine needs to break the matching of a subject into matching portions, each of them corresponding to the matching of a subexpression. It is then possible to determine how each of these matchings was found and retrieve their results.

Captures are very useful, but they modify the matching process to match in a deterministic way, and can disturb some regular expression properties. The POSIX rules to achieve determinism [IEEE 2004], for instance, alter the associative property of concatenation [Fowler 2003].

In all pattern-matching implementations, parentheses are used for making captures: the expression `(p)` indicates that the string that matched with pattern `p` will be captured for later retrieval. The pattern `"([a-z]*)"` is a simple example that matches strings enclosed in double quotes and captures these strings.

In Perl's, Python's and Ruby's implementations, it is possible to make a capture inside a positive lookahead. As an example, if the pattern `p(?=(q))` matches, it produces the value captured by `(q)`. Captures inside negative lookaheads are ignored, but the libraries documentation do not describe this behavior.

2.5. Backreferences

Backreferences are used to reference a captured value at matching time. Backreferences have the syntax `\n`, where `n` is the capture identifier. The backreference `\1` in pattern `([a-z])\1` is replaced with the substring matched by the capture `([a-z])`.

Backreferences extend regular expressions by allowing the recognition of more than regular languages. As an example, the expression `(a+)(b+)\1\2` defines the language $\{\ a^i b^j a^i b^j \mid i, j > 0\ \}$ which is not context-free.

Backreferences can result in an exponential running time of the matching algorithm. This problem is unavoidable because the 3-SAT problem can be reduced to regexes with backreferences. Therefore, any matcher thus constructed is NP-Complete [Abigail].

The documentation of Perl, Python and Ruby regexes show some simple examples of backreferences, but do not explain their behavior when combined with other constructions. Some questions are difficult to answer without testing. For example: can we quantify a backreference? Can backreferences be used inside character classes? Can we capture backreferences? When testing these combinations we observe that backreferences cannot be used inside character classes, but only Python's documentation describes this

behavior. Our tests also showed that a backreference can be quantified in all implementations and can also can be captured, as in `(ab)(\1)`.

2.6. Lookbehind

Lookbehinds are used to check whether a pattern is preceded by another, without consuming any input. This construction is available in Perl, PCRE and Python implementations; Ruby does not offer lookbehinds. The syntax for a positive lookbehind is `(?<=p)` and for a negative lookbehind is `(?<!p)`.

A lookbehind checks if the portion of the subject that precedes the current matching position matches a given pattern. To perform this check, the matching engine saves the current position and moves back, trying to match the pattern. After that, the engine resumes the matching process from the saved position, and tries to match the rest of the expression. As other zero-width assertives, lookbehinds do not consume any input.

Perl and Python regexes documentation state that lookbehinds can only use expressions that match "fixed width" strings. This restriction allows the pattern-matching engine to know how many matching positions it needs to go back. However, the documentation do not clearly specify what kind of expressions can be accepted. A closer inspection of these regex implementations reveals that expressions like `abc`, `a{2}` and `ab|cd` are accepted but not `a*`, `a+`, `a{3,4}` or `ab|c`. They also exclude backreferences inside lookbehinds because backreferences are patterns that match a variable number of characters. The PCRE implementation permits more types of expressions inside lookbehinds, such as `a|ab`.

Some other pattern-matching implementations are less restrictive with respect to expressions inside lookbehinds [Friedl 2006]. Java's lookbehind allows patterns like `(ab?)`, which matches an `"a"` optionally followed by a `"b"` (a matching with one or two characters). The pattern-matching engine of Microsoft's .NET does not impose any restriction on expressions inside lookbehinds. However, variable length patterns inside lookbehinds can result in efficiency problems when used without precautions, specially for matchings near the end of a large input, because the pattern-matching machine may need to try a matching from the beginning of the subject [Friedl 2006].

3. Parsing Expression Grammars

Parsing Expression Grammars (PEGs) are a formal system for language recognition based on *Top Down Parsing Languages* [Ford 2004]. A PEG is an alternative for Chomsky's generative system of grammars, particularly context-free grammars and regular expressions. PEGs have also been shown to be powerful enough to express all deterministic *LR(k)* languages and even some non-context free languages [Ford 2004].

A PEG consists of a set of rules of the form $\mathtt{A} \leftarrow e$, where `A` is a nonterminal and e is a *parsing expression*, plus an initial parsing expression e_s. Table 2 summarizes the operators for constructing these expressions.

In an ordered choice expression e_1 / e_2, the option e_2 is only tried if e_1 fails; once an alternative has been chosen, it cannot be changed because of a later failure. This means that a PEG does only *local* backtracking. As an example, consider the following PEG:

Operation	Description
`''`	literal
`""`	literal
`[...]`	character class
.	any character
e?	optional
e*	zero-or-more
e+	one-or-more
&e	and-predicate
!e	not-predicate
e_1 e_2	concatenation
e_1 / e_2	ordered choice

Table 2. Operators for constructing parsing expressions

$$\begin{aligned} S &\leftarrow A\ B \\ A &\leftarrow p_1\ /\ p_2\ /\ \ldots\ /p_n \end{aligned}$$

The attempt to match `S` begins by trying to match `A`. To match `A`, a match for its first alternative, p_1, is tried; if it fails, a match for p_2 is tried, and so on. If a pattern p_i matches, there is no backtracking for `A` even if the subsequent match for `B` fails.

PEGs also offer two *syntactic predicates*, which do not consume any input. The *not* predicate, denoted by `!`, corresponds to a negative lookahead: the expression !e matches only if pattern e fails with the current subject. The *and* predicate, denoted by `&`, corresponds to a positive lookahead. The expression `&`e is defined as `!!`e: it matches only if e suceeds.

As an example, the following PEG matches C comments:

```
C ← "/*" (!"*/" .)* "*/"
```

After a match of the comment's start (`"/*"`), the internal expression matches the comment's text by repeatedly consuming characters as long as `"*/"` (the comment's end) does not match.

The following PEG recognizes simple arithmetic expressions:

```
   Exp ← Factor (FactorOp Factor)* !.
Factor ← Term (TermOp Term)*
  Term ← "-"? Number
FactorOp ← [+-]
TermOp ← [*/]
Number ← [0-9]+
```

An arithmetic expression (`Exp`) is a sequence of one or more factors separated by an operator (`"+"` or `"-"`). The pattern `!.` means that this sequence cannot be followed by

any character. A factor is a sequence of one or more terms separated by an operator (`"*"` or `"/"`). A term is a number optionally prefixed with a `"-"`. A number is a sequence of one or more digits.

In particular, the PEG `""` (an empty parsing expression) always match an empty prefix of any string. The PEG `!""`, on the other hand, fails for any input. As we will see, the empty PEG has an important role in our conversion algorithm.

4. Converting regular expressions to PEGs

Despite the syntactic similarity between regular expressions and PEGs, a PEG with a similar syntax to a regular expression does not usually recognize the same language defined by the expression. As an example, the regular expression `(a|ab)c` defines the language {*ac, abc*}, but the PEG `(a/ab)c` does not accept the string `"abc"`. This happens because after matching the first alternative `a` with `"a"`, the PEG engine tries to match `c` with the rest of the input, `"bc"`. Although this subsequent match fails, there will be no backtracking for the second alternative of the ordered choice, `ab`.

The example above is a particular case of the structure $(p_1 \mid p_2)\, p_3$. In a PEG, if p_1 matches a portion of the subject and p_3 does not match the rest of the input, there will be no backtracking for the alternative p_2. In pattern-matching engines, this backtracking always occurs. In order to build a PEG that recognizes the language defined by a regular expression with such an structure, we need to append the pattern that follows the alternation to the end of each alternative of the PEG ordered choice. This will result in a expression like $(p_1 p_3 \mid p_2 p_3)$.

In the following subsections we present an algorithm that correctly converts regular expressions and some of their extensions into PEGS. The basic idea of this algorithm is to introduce an explicit *continuation* to guide the conversion process. As an example, let us return to the expression `(a|ab)c`, which is a concatenation of two subexpressions: `(a|ab)` and `c`. The second subexpression, `c`, can be seen as the continuation of the first subexpression, `(a|ab)`, in the sense that it defines what needs to be matched subsequently. When we concatenate this continuation to all the alternatives in the first subexpression, we obtain the PEG `ac/abc`, which correctly recognizes the language {*ac, abc*}.

4.1. Continuation-based conversion

The *continuation-based conversion* is a function $\Pi(e,\ k)$ that receives a regular expression (*e*) and a continuation (*k*), and returns a parsing expression. The continuation defines what needs to be matched after *e*. As a side-effect, throughout the conversion process a PEG is built such that at the end of the conversion the resulting PEG is equivalent to the given regular expression.

Note that the continuation-based conversion process begins with the empty parsing expression `""` as continuation, because when the entire expression matches, there is nothing else to match.

We will define Π by cases. The first four cases are as follows:

$$\Pi(\varepsilon,\ k) = k \tag{1}$$

$$\Pi(\mathtt{c},\, k) = \mathtt{c}\, k \tag{2}$$

$$\Pi(e_1 e_2,\, k) = \Pi(e_1,\, \Pi(e_2,\, k)) \tag{3}$$

$$\Pi(e_1 | e_2,\, k) = \Pi(e_1,\, k) \,/\, \Pi(e_2,\, k) \tag{4}$$

Case 1 describes the conversion of the regular expression ε, which matches the empty string. This conversion results in the parsing expression provided as the continuation.

Case 2 describes the conversion of a single character ($\mathtt{c}$). Its result is the parsing expression $\mathtt{c}k$, that matches $\mathtt{c}$ and then k.

Case 3 describes concatenation. To convert a concatenation $e_1 e_2$, we begin by converting the second subexpression using the original continuation. This conversion is denoted by $\Pi(e_2,\, k)$. The resulting parsing expression is then used as the continuation for converting the first subexpression.

Case 4 shows that to convert an alternation, we need to convert each alternative using the original continuation k. This is the key step for distributing the concatenation to the alternatives, providing the solution to our original problem, the conversion of expressions like $(p_1 \,|\, p_2)\, p_3$ into $(p_1\, p_3 \,|\, p_2\, p_3)$.

To deal with the conversion of repetitions e^*, we need something slightly more complex. Our insight for this conversion is that a repetition can be defined as follows:

$$e^* \;= e\; e^* \,|\, \varepsilon$$

Using this equality, we can expand $\Pi(e^*,\, k)$:

$$\begin{aligned}
\Pi(e^*,\, k) &= \Pi(e\, e^* \,|\, \varepsilon,\, k) \\
&= \Pi(e\, e^*,\, k) \,/\, \Pi(\varepsilon,\, k) && \text{(case 4)} \\
&= \Pi(e\, e^*,\, k) \,/\, k && \text{(case 1)} \\
&= \Pi(e\,, \Pi(e^*,\, k)) \,/\, k && \text{(case 3)}
\end{aligned}$$

Now, if we substitute $\Pi(e^*,\, k)$ for a nonterminal $\mathtt{A}$, we obtain

$$\mathtt{A} = \Pi(e,\, \mathtt{A}) \,/\, k$$

We then add the nonterminal $\mathtt{A}$ and its corresponding rule to the PEG built by the conversion process, and define the conversion of a repetition as follows:

$$\begin{aligned}
\Pi(e^*,\, k) \;&=\; A \\
\mathit{where}\;\; A \;&\leftarrow\; \Pi(e,\, A) \,/\, k
\end{aligned} \tag{5}$$

As an example of the conversion of a regular expression into a PEG, let us convert the expression `(ba|a)*a`:

$$\begin{aligned} \Pi(\texttt{(ba|a)*a}, \texttt{""}) &= \Pi(\texttt{(ba|a)*}, \Pi(\texttt{a}, \texttt{""})) \\ &= \Pi(\texttt{(ba|a)*}, \texttt{a}) \\ &= \texttt{A} \\ \texttt{A} &\leftarrow \texttt{baA} \,/\, \texttt{aA} \,/\, \texttt{a} \end{aligned}$$

Note that `baA/aA` is the result of $\Pi($`(ba|a)`$,$`A`$)$.

Applying the continuation-based conversion to sequences of alternations may result in an exponencial growth of the size of the resulting parsing expressions. As an example, the expression `(a|b)(c|d)` produces the PEG `ac/ad/bc/bd`, doubling the number of alternatives. However, this does not mean that the resulting PEG will double the time required to match the original expression using a regex engine. Because function Π merely converts regex global backtrackings to PEG local backtrackings, the number of backtrackings involved when matching a subject with either the original expression or the produced PEG is actually the same.

An equivalent conversion to case 4 that preserves the number of alternatives in the original expression and, thus, reduces space requirements, is given below:

$$\begin{aligned} \Pi(e_1|e_2,\, k) &= \Pi(e_1,\, \texttt{A}) \,/\, \Pi(e_2,\, \texttt{A}) \\ \texttt{A} &\leftarrow k \end{aligned}$$

However, except for some uncommon examples, like the 82-line Perl expression used for validating email adresses [Warren 2002], most practical uses of regexes involve relatively small expressions. The largest sequence of alternations shown in the book *Mastering Regular Expressions* [Friedl 2006], for instance, contains eight alternatives. All the other examples in the book contain four alternatives at most.

A formal proof of the correctness of the continuation-based conversion is presented in Medeiros' doctoral thesis [Medeiros 2010]. The basic idea of this proof is to show that if a regular expression e_k is equivalent to a PEG p_k, denoted $e_k \sim p_k$, then the concatenation of an expression e to e_k is equivalent to the conversion of e using p_k as a continuation: $e_k \sim p_k \Rightarrow e\, e_k \sim \Pi(e, p_k)$.

4.2. Converting regexes to PEGs

We will now discuss how to convert some of the regex extensions presented earlier.

The conversion of lazy quantifiers is similar to the conversion of greedy quantifiers, described in the previous section. In order to make the PEG simulate the behavior of a lazy repetition, we only need to change the order of the alternatives in the rule that we associate to the nonterminal. Case 6 describes how to do it:

$$\begin{array}{rcl} \Pi(e*?,\, k) & = & \texttt{A} \\ where\ \texttt{A} & \leftarrow & k \,/\, \Pi(e,\, \texttt{A}) \end{array} \tag{6}$$

With this new definition, `A` first tries to match only the continuation; if this fails, it will match e and try again.

Independent subexpressions, as described in section 2, match independently of their enclosing expressions. The basic concept here is to disallow backtrackings that retry to match portions already consumed. Note that this is the original behavior of backtracking in PEGs, so all we have to do is to avoid converting e with the given continuation. Instead, we convert e using the empty PEG as continuation and concatenate the original continuation to the result of this conversion, as described by case 7:

$$\Pi((?{>}e),\, k) = \Pi(e,\, \texttt{""})\, k \tag{7}$$

We have discussed earlier that a possessive quantifier is a particular case of an independent expression. Therefore, the conversion of possessive quantifiers is straightforward if we use case 7 as its basis:

$$\Pi(e*+,\, k) = \Pi(e*,\, \texttt{""})\ k \tag{8}$$

A subexpression used in a lookahead also matches independently of the rest of the pattern. To convert an expression in a lookahead we thus use the empty PEG as continuation, as we have done for converting independent subexpressions. The resulting expression is then prefixed with a syntactic predicate so that it does not consume any input. The conversion of a positive lookahead uses the *and* predicate, as described by case 9:

$$\Pi((?=e),\, k) = \&\Pi(e,\, \texttt{""})\, k \tag{9}$$

To convert a negative lookahead, we use the *not* predicate (`!`):

$$\Pi((?!\, e),\, k) = !\, \Pi(e,\, \texttt{""})\, k \tag{10}$$

Captures pose an insteresting problem, because our conversion algorithm may break up a capture. To illustrate this problem, let us consider the expression `{a(b|c)}d`. To avoid confusion, we have denoted the start and the end of a capture with the metacharacters `'{'` and `'}'`. Our algorithm would distribute the capture delimiters to the alternatives, resulting in a PEG like `{a(b}d|c}d)`, where the capture delimiters are not statically balanced. So, clearly captures cannot be converted to equivalent captures in PEGs.

5. Conclusions

This work presented a new algorithm to convert regexes into PEGs. It introduced the continuation-based conversion function, showing how to use it to convert pure regular expressions into PEGs; it then presented some extensions of this function that cover some common constructions of Perl's regex such as lookaheads, independent expressions, and lazy and possessive quantifiers.

The main contribution of this work is to permit the execution of regexes on the formal model of PEGs. Besides providing a better understanding of the semantics of regexes, the conversion of regexes into PEGs also allows us to benefit from the performance model of PEGs.

As we discussed earlier, captures pose interesting problems, and cannot be converted to equivalent captures in PEGs. LPeg [Ierusalimschy 2008], a PEG implementation, provides several powerful capture functions. Based on this implementation, we are developing an alternative for converting regex captures.

Backreferences also cannot be converted into PEGs. This regex extension is directly related to captures and thus its conversion depends on the provision of this facility.

The lookbehind construction also poses interesting problems. Behind its apparent simplicity, it implies changes in the conceptual model supporting PEGs, which assumes that only the rest of the input may affect the match. In PEGs, it can create subtle cases of left recursion and infinite loops. We are currently investigating some alternatives for converting this construction.

Based on the observation that right-linear grammars have the same behavior seen both as PEGs and as CFGs, one of the authors has already proposed a translation from finite automata to PEGs. The basic idea of this translation is to use traditional techniques to convert finite automata into right-linear grammars, producing a PEG that recognizes the same language as the given automata. In this paper we presented an algorithm to directly convert regular expressions and some regex extensions into PEGs; as far as we know, this is the first proposal of an algorithm to perform such a conversion.

References

Abigail. Reduction of 3-CNF-SAT to Perl Regular Expression Matching. `http://perl.plover.com/NPC/NPC-3SAT.html`, visited on May 2010.

Ford, B. (2004). Parsing Expression Grammars: a recognition-based syntactic foundation. In *POPL '04: Proceedings of the 31st Symposium on Principles of Programming Languages*, pages 111–122, New York, NY. ACM.

Fowler, G. (2003). An Interpretation of the POSIX regex Standard. `http://www2.research.att.com/~gsf/testregex/re-interpretation.html`, visited on May 2010.

Friedl, J. (2006). *Mastering Regular Expressions*. O'Reilly Media, Inc.

Habibi, M. (2004). *Real World Regular Expressions with Java 1.4*. APress.

Hazel, P. (2009). PCRE - Perl Compatible Regular Expressions. `http://www.pcre.org/`, visited on January 2010.

IEEE (2004). The Open Group Base Specifications Issue 6. http://www.opengroup.org/onlinepubs/009695399/basedefs/xbd_chap09.html , visited on December, 2009.

Ierusalimschy, R. (2008). Parsing Expression Grammars for Lua. http://www.inf.puc-rio.br/~roberto/lpeg/, visited on November 2009.

Ierusalimschy, R. (2009). A text pattern-matching tool based on Parsing Expression Grammars. *Softw. Pract. Exper.*, 39(3):221–258.

Lutz, M. (2006). *Programming Python*. O'Reilly Media, Inc.

Medeiros, S. (2010). *Um Estudo Sobre Gramáticas de Expressões de Parsing e a sua Correspondência com Expressões Regulares e Gramáticas Livres de Contexto $LL(k)$-Forte*. PhD thesis, PUC-Rio.

Thomas, D., Fowler, C., and Hunt, A. (2009). *Programming Ruby 1.9: The Pragmatic Programmers' Guide*. Pragmatic Bookshelf.

Truskett, I. Perl - Regular Expressions Reference. http://perldoc.perl.org/perlreref.html, visited on January 2010.

Warren, P. (2002). regexp-based address validation. http://www.ex-parrot.com/~pdw/Mail-RFC822-Address.html, visited on July 2010.