SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2009; **39**:221–258
Published online 17 July 2008 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe.892

A text pattern-matching tool based on Parsing Expression Grammars

Roberto Ierusalimschy*, †

PUC-Rio, Rio de Janeiro, Brazil

SUMMARY

Current text pattern-matching tools are based on regular expressions. However, pure regular expressions have proven too weak a formalism for the task: many interesting patterns either are difficult to describe or cannot be described by regular expressions. Moreover, the inherent non-determinism of regular expressions does not fit the need to capture specific parts of a match. Motivated by these reasons, most scripting languages nowadays use pattern-matching tools that extend the original regular-expression formalism with a set of *ad hoc* features, such as greedy repetitions, lazy repetitions, possessive repetitions, 'longest-match rule,' lookahead, etc. These *ad hoc* extensions bring their own set of problems, such as lack of a formal foundation and complex implementations. In this paper, we propose the use of Parsing Expression Grammars (PEGs) as a basis for pattern matching. Following this proposal, we present LPEG, a pattern-matching tool based on PEGs for the Lua scripting language. LPEG unifies the ease of use of pattern-matching tools with the full expressive power of PEGs. Because of this expressive power, it can avoid the myriad of *ad hoc* constructions present in several current pattern-matching tools. We also present a Parsing Machine that allows a small and efficient implementation of PEGs for pattern matching.

Received 1 October 2007; Revised 20 May 2008; Accepted 21 May 2008

KEY WORDS: pattern matching; Parsing Expression Grammars; scripting languages

1. INTRODUCTION

Pattern-matching facilities are among the most important features of modern scripting languages, such as Perl, Python, Lua, Ruby, and PHP.

One of the basic foundations for pattern matching was regular expressions [1,2]. Early Unix tools, such as `grep` and `ed`, used the full theory of automata in their implementation: they took a regular expression, translated it to a non-deterministic automaton, and then did a multiple-state

*Correspondence to: Roberto Ierusalimschy, PUC-Rio, Rua M. S. Vicente 225, Rio de Janeiro, 22451-900, Brazil.
†E-mail: roberto@inf.puc-rio.br

Contract/grant sponsor: Brazilian Research Council (CNPq); contract/grant number: 300993/2005-6

simulation of a deterministic automaton, which could recognize strings in the given language very efficiently. However, regular expressions have several limitations as a tool for pattern matching, even for basic tasks. Because they have no complement operator, some apparently simple languages can be surprisingly difficult to define. Typical examples include C comments, which is trickier than it seems, and C identifiers, which must exclude reserved words such as `for` and `int`.

Another limitation of the original theory of regular expressions is that it concerns only the recognition of a string, with no regard for its internal structure. Several common applications of pattern matching need to break the match in parts—usually called *captures*—to be manipulated. There are some proposals to implement this facility within DFAs, such as Laurikari's TRE library [3,4], but most current tools based on formal regular expressions do not support this facility. Moreover, captures may subtly disturb some properties of regular expressions; for instance, the definition of captures in POSIX regexes makes concatenation no longer associative [5].

Captures also demand that the pattern matches in a deterministic way with a given subject. The most common disambiguation rule is the *longest-match rule*, used by tools from Lex to POSIX regexes, which states that each sub-pattern should match in the longest possible way. However, as argued by Clarke and Cormack [6], any search algorithm using the longest-match rule may need to traverse the subject multiple times.

Confronted by these difficulties, several pattern-matching implementations have forsaken regular expressions as their formal basis, despite the fact that they still call the resulting patterns 'regular expressions.' This category includes POSIX regexes, Perl regexes, *Perl compatible regular expressions* (PCRE, used by Python), and others. However, these new libraries bring their own set of problems:

- They have no formal basis. Instead, they are an aggregate of features, each focused on a particular regular-expression limitation. There are particular constructions to match word frontiers, lazy and possessive repetitions, look ahead, look behind, non-backtracking sub-patterns, etc. [7].
- Because they have no formal basis, it is difficult to infer the meaning of some exotic combinations, such as the use of captures inside negative lookaheads or substitutions on patterns that can match the empty string. (How many times does the empty string match against the empty string?)
- Their implementations use backtracking, which makes several innocent-looking patterns require non-linear time. For instance, consider that we want to break a line composed of five fields separated by commas and ending with a dot or a semicolon. A simple pattern in Perl for this task could look like this one:

  ```
  (.*),(.*),(.*),(.*),(.*)[.;]
  ```

  Although this pattern is efficient for well-formed lines, it takes 27 seconds to match against a ill-formed line consisting of 80 commas[‡].
- These implementations have no performance models; hence, it is hard to predict the performance of any particular pattern. For instance, if we make a small change in the previous pattern substituting a `;` for the final `[.;]`, it becomes as efficient for ill-formed lines as for well-formed ones.

[‡]Test ran with Perl v5.8.8 on a Pentium 4 2.93 GHz.

DOI: 10.1002/spe

- They still lack the expressive power to express many useful non-regular patterns, such as strings with balanced parentheses or nested comments.

In this paper, we present LPEG, a new pattern-matching tool based on *Parsing Expression Grammars* (PEGs) [8] for the Lua scripting language. PEGs are a revival of *Generalized Top-Down Parsing Languages* (GTDPL), an old theory of formal grammars with an emphasis on parsing instead of language generation, using restricted backtracking [9]. Although a PEG looks suspiciously similar to a Context-Free Grammar (CFG), it does not define a language, but rather an algorithm to recognize a language.

One of the main advantages advocated for PEGs is that they can express a language syntax down to the character level, without the need for a separate lexical analysis [10]. Here, we take a complementary view, showing that PEGs provide a solid base for pattern matching, which is powerful enough to express syntactic structures when necessary.

PEGs have several interesting features for pattern matching:

- PEGs have an underlying theory that formally describes the behavior of any pattern.
- PEGs implement naturally several pattern-matching mechanisms, such as greedy repetitions, non-greedy repetitions, positive lookaheads, and negative lookaheads, without *ad hoc* extensions.
- PEGs can express all deterministic $LR(k)$ languages (although not all deterministic $LR(k)$ *grammars*), and even some non-context-free languages [8].
- PEGs allow a fast and simple implementation based on a *parsing machine*, which we will describe in Section 4.
- Although it is possible to write patterns with exponential time complexity for the parsing machine[§], they are much less common than in regexes, thanks to the limited backtracking. In particular, patterns written without grammatical rules always have a worst-case time $O(n^k)$ (and space $O(k)$, which is constant for a given pattern), where k is the pattern's star height. Moreover, the parsing machine has a simple and clear performance model that allows programmers to understand and predict the time complexity of their patterns. The model also provides a firm basis for pattern optimizations.
- It is easy to extend the parsing machine with new constructions without breaking its properties. For instance, adding back references to the machine does not make the problem NP-complete[¶].

To make PEG more suitable for pattern matching, LPEG introduces three main novelties. First, it modifies slightly the original PEG syntax. Instead of grammars, expressions are the main concept for LPEG; grammars are used only when needed. Second, it unifies the concepts of captures, semantic actions, and substitutions. It is as easy to use LPEG to break a date into day/month/year as to create a syntactical tree for a piece of code. Moreover, LPEG implements substitution, a common task for pattern-matching tools, as a new form of capture, instead of as a separate function. Third, as we already mentioned, LPEG uses a novel parsing machine to implement PEGs.

[§] Actually, there is an algorithm that matches any PEG in linear time with the input size [9], but it also needs linear space; hence, we do not use it in LPEG. In Section 4 we will discuss these implementation aspects.

[¶] Regular expressions augmented with back references, such as PCRE, can solve the 3-satisfiability problem, which means that any matcher for such a system is NP-complete [11].

DOI: 10.1002/spe

In the following section we will discuss PEGs in more detail. In Section 3 we present LPEG, the pattern-matching tool for Lua based on PEGs. Section 4 describes in detail our parsing machine, LPEG's matching engine. In Section 5 we assess the performance of the parsing machine with an analytical result and some benchmarks against other pattern-matching tools. Finally, in the last section we draw some conclusions.

2. PARSING EXPRESSION GRAMMARS

PEGs are a formal system for language recognition based on *Top-Down Parsing Languages* (TDPL) [8].

At first glance, PEGs look quite similar to CFGs extended with some regular-expression operators. The following PEG is a self-description of the original PEG syntax, down to the character level:

```
grammar     <- (nonterminal '<-' sp pattern)+
pattern     <- alternative ('/' sp alternative)*
alternative <- ([!&]? sp suffix)+
suffix      <- primary ([*+?] sp)*
primary     <- '(' sp pattern ')' sp / '.' sp / literal /
               charclass / nonterminal !'<-'
literal     <- ['] (!['] .)* ['] sp
charclass   <- '[' (!']' (. '-' . / .))* ']' sp
nonterminal <- [a-zA-Z]+ sp
sp          <- [ \t\n]*
```

Like a BNF, a PEG is written as a sequence of one or more rules. A *pattern*, the right-hand side of each rule, is a sequence of one or more alternatives separated by a slash (`/`); this slash is the *ordered choice* operator, comparable to the vertical bar (`|`) used in BNF. Each alternative is a sequence of suffix expressions optionally prefixed with a predicate operator, which we will describe later.

Parentheses are used for grouping. As in regular expressions, a dot matches any single character. Each suffix expression may be followed by suffix operators: E`*` means zero or more E's, E`+` means one or more E's, and E`?` means an optional E. Unlike regular expressions, literals must be quoted; this requirement avoids complications with 'magic' characters and allows us to use spaces to better format a grammar. Again like regular expressions, a string enclosed in square brackets represents a *character class*: it matches any character of the string; inside a character class, x`-`y means any character between x and y. For instance, `['] ` matches only a quote, and `[a-z_]` matches any lower-case letter or an underscore.

Despite the syntactic similarity, the semantics of a PEG is different from the semantics of a CFG. While a CFG describes a language, a PEG describes a *parser* for a language[‖]. More specifically, a PEG describes a top-down parser with restricted backtracking [9].

The ordered choice operator makes the result of any parsing unambiguous: in a choice `A / B`, `B` is only tried if `A` fails. Given a PEG and an input string, either the parsing fails or it accepts a unique prefix of the input string. For complex grammars, such as those describing complete programming

[‖]This is why PEGs use a left arrow `<-` instead of the right arrow commonly used in CFGs.

DOI: 10.1002/spe

languages, reliance on ordered choice to solve ambiguities may lead to unexpected behavior. But for simple grammars ordered choice is a powerful tool, as we will see.

Restricted backtracking means that a PEG does only *local* backtracking, that is, it only backtracks while choosing an appropriate option in a rule. Once an option has been chosen, it cannot be changed because of a later failure. As an example, consider the following grammar fragment, where each E_i is an arbitrary expression:

```
S <- A B
A <- E1 / E2 / E3
B <- ...
```

When trying to match the input string against `S`, the parser starts with `A`. To match `A`, the parser will first try the expression E_1. If that match fails, it will backtrack and try E_2, and so on. Once a matching option is found, however, there is no more backtracking for this rule. If, after choosing E_i as the option for `A`, the subsequent match of `B` fails, the entire sequence `A B` fails. A failure in `B` will not cause the parser to try another option for `A`; that is, the parser does not do *global* backtracking.

This difference in semantics has several important consequences. Among other things, it allows the grammar to formally specify the kind of repetition it wants, without extra constructions and arbitrary conventions like 'the longest-match rule.' Repetitions in PEG may be greedy or non-greedy, and blind or non-blind.

2.1. Repetitions

A blind greedy repetition, sometimes called *possessive*, always matches the maximum possible span (therefore greedy), disregarding what comes afterwards (therefore blind). Frequently this is exactly what we want, but several pattern-matching tools do not provide this option. For instance, when matching `[a-z]*'1'` against the string `"count2"`, it is useless to try a shorter match for the `[a-z]*` part after the `'2'` fails against `'1'`. Nevertheless, this is what several backtracking tools do. This behavior is the cause of the combinatorial time for the failure of patterns such as `(.*),(.*),(.*),(.*),(.*)[.;]`, which we saw in the previous section.

In PEGs, a repetition E`*` is equivalent to the following rule:

```
S <- E S / ''
```

Ordered choice implies that this repetition is greedy: it always tries first to repeat. Local backtracking implies that it is blind: a subsequent failure will not change this longest matching. Hence, blind greedy repetition is the norm.

A non-blind greedy repetition is one that repeats as many times as possible (therefore greedy), as long as the rest of the pattern matches (therefore non-blind). It is the norm in conventional pattern-matching tools. This kind of repetition often implies some form of backtracking. When you write a pattern like `.*`E, the matcher must find the *last* occurrence of E in the subject. Of course, to know that the occurrence is the last one the matcher must go until the subject's end, no matter the algorithm.

If you need non-blind greedy repetition, you can write it with a PEG like this:

```
S <- E1 S / E2
```


DOI: 10.1002/spe

This pattern will always try first the option E_1 `S`, which consumes one instance of E_1 and tries `S` again. This will go on until the subject's end. Once there, both E_1 and E_2 fails; it cannot match an `S` anymore and hence the previous E_1 `S` fails, too. The matcher then tries to match E_2 at that previous position. If that also fails, it backtracks again, and again tries to match E_2. Hence, it backtracks step by step until it finds a E_2 or until there is no more backtracking and the entire pattern fails. As an example, the following grammar matches a subject up to its last digit:

```
S <- . S / [0-9]
```

Blind non-greedy repetition is not very useful—it always matches the empty string. Non-blind non-greedy repetition (also called *lazy* or *reluctant*), on the other hand, may be convenient. Several regex engines provide special operators for this kind of repetition; in Perl, for instance, we write E`*?` for a non-greedy repetition. PEGs do not need any special mechanism for it. If you want to match the minimum number of E_1 up to the first E_2, you can write the following:

```
S <- E2 / E1 S
```

This pattern first tries to match E_2. If that fails, it matches E_1 and repeats. As a concrete example, the following grammar matches C comments:

```
Comment <- '/*' Close
Close   <- '*/' / . Close
```

2.2. Other constructions

Besides what we have seen so far, PEGs offer two other operators, called *syntactic predicates*. The *not* predicate corresponds to a negative look ahead. The expression `!`E matches only if the pattern E fails with the current subject. Because in any case either e fails or `!`E fails, this pattern never consumes any input; its result is simply success or fail. As an example, PEGs do not need a special pattern (usually denoted by `$` in regex engines) to match the subject's end. The pattern `!.` succeeds only if `.` (which matches any character) fails; this fail can only happen at the end of the subject, when there are no more characters available.

The second syntactic predicate is the *and* predicate, denoted by a prefixed `&`. It corresponds to the positive look-ahead operation. The expression `&`E is defined as `!!`E: it succeeds only when E succeeds, but does not consume any input.

Syntactic predicates greatly expand the expressiveness of PEGs: they provide a strong look-ahead mechanism, and the *not* predicate provides a kind of a complement operator. For instance, to match a lower-case identifier different from `int` or `float`, the following pattern may be used:

```
!(('int' / 'float') ![a-z]) [a-z]+
```

The pattern `(('int' / 'float') ![a-z])` succeeds when it matches `int` or `float` not followed by a lower-case letter (that is, as a complete word). Hence, its negation succeeds when the current position does not match any of those words. After this check, the pattern `[a-z]+` matches an identifier. This is a typical example of a tricky pattern to express with strict regular expressions.

The *not* predicate can replace non-greedy repetitions in most situations. Instead of writing `E*?B` (assuming a Perl-like `*?` non-greedy qualifier), we can write `(!B E)* B`, which reads as 'matches instances of `E` as long as `B` does not match, and then matches `B`.' Using this technique, we can rewrite our grammar for C comments like this:

```
Comment <- '/*' (!'*/' .)* '*/'
```


DOI: 10.1002/spe

PEGs do not support left recursion. For instance, if we write a grammar like

```
S <- S X / ...
```

the result is an infinite loop. We say that such grammars are not *well formed* [8]. Similarly, loops of patterns that can match the empty string (e.g. `''*`) are not well formed, as their expansions result in left-recursive rules. Although the general problem of whether a grammar is well formed is undecidable [8], some PEG tools implement conservative algorithms to detect these degenerate loops.

PEGs present several similarities with parsing combinators [12]. Both use ordered choice and has an interface based on the acceptance of input prefixes. Moreover, both share the restrictions common to all top-down parsers, such as no support for left recursion.

Typically, parsing combinators work either deterministically, by using only the first result from a parsing function, or with full backtracking. However, it is easy to construct combinators that implement restricted backtracking as in PEG. Such combinators do not need to return lists of results, and as such are more friendly to strict (non-lazy) languages. In fact, our first attempt to implement LPEG was based on parsing combinators. The main problem with parsing combinators, for us, is that the resulting parsers are black boxes: because a parser is represented by a function, we cannot inspect its internal structure. This opacity inhibits the construction of tools to optimize parsers, check for (invalid) left recursion, and other similar operations.

2.3. Right-linear grammars

PEGs have a interesting relationship with right-linear grammars. A *right-linear grammar* is a grammar in which each production alternative has at most one non-terminal, at its end. Because there are no sequences like `A B` (with two non-terminal symbols), the question of whether the parser tries another option for `A` if `B` fails does not arise. Therefore, these grammars have the same behavior both as PEGs and as CFGs. In particular, we can use traditional techniques to translate a finite automata into a PEG. For instance, the NFA in Figure 1 would result in the following grammar:

```
A <- . A / 'a' B
B <- 'n' C
C <- 'a' D
D <- ''
```

Depending on the order of the terms in rule `A`, the grammar matches the longest or the shortest subject's prefix ending with `'ana'`.

Often it is easier to write an automata for a language than an equivalent regular expression. An automatic translation from the automata to a regular expression usually results in an expression too

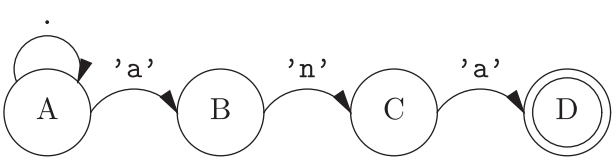

Figure 1. A NFA that accepts strings ending with `'ana'`.

DOI: 10.1002/spe

big and unwieldy to be useful. Using PEGs, we can translate the automata automatically to a usable grammar. As a particular example, the following grammar recognizes strings with an even number of 0's and an even number of 1's:

```
EE <- '0' OE / '1' EO / !.
OE <- '0' EE / '1' OO
EO <- '0' OO / '1' EE
OO <- '0' EO / '1' OE
```

(`EE` means 'Even zeros and Even ones'.) I invite the reader to express this regular language with regular expressions.

3. THE LPEG LIBRARY

The LPEG library is an implementation of PEGs for the Lua language.

Unlike other PEG implementations, which aim at parsing, LPEG aims at pattern matching. Therefore, it turns PEG inside out: while PEGs define grammars using pattern expressions as an auxiliary construction, in LPEG the main construction is the pattern, and grammars are only a particular way to create patterns. More formally, if we consider the previous grammar we presented for PEGs, the grammar for LPEG would be like this:

```
pattern     <- grammar / simplepatt
grammar     <- (nonterminal '<-' sp simplepatt)+
simplepatt  <- alternative ('/' sp alternative)*
alternative <- ... (as before)
```

The initial symbol now is `pattern`, which may be a whole grammar or a simple pattern. Hence, LPEG still accepts complete grammars, as the original PEG. However, it also accepts something as simple as `[a-z]+`, which should be written as `S <- [a-z]+` to be accepted as a PEG. Moreover, remember that one of the options for a `primary` is a parenthesized pattern, so that a parenthesized grammar may be used as part of a larger pattern.

LPEG offers two modules, each with its own modus operandi. The first one, the `re` module, has a modus operandi similar to other pattern-matching tools, where patterns are described by strings written in a little language based on regular-expression syntax. The second module, `lpeg`, follows the SNOBOL tradition, where patterns are first-class values built with regular language constructions.

Before seeing more details about these two modules, however, we will see a brief overview of Lua.

3.1. A brief overview of Lua

Here we will review a few concepts of Lua that we need in this paper. For more information about the language, please refer to its documentation [13,14].

Like other scripting languages, Lua is dynamically typed. Variables do not have types; each value carries its own type, which is checked when needed, at run time. It is very easy to integrate code written in Lua with code written in C. Several libraries for Lua are written in C.

DOI: 10.1002/spe

Strings in Lua are immutable values. We can write a literal string enclosing its content in single or double quotes. The only difference between quotes is that we can use the other quote inside the string:

```
print('hello "world"')   --> hello "world"
print("hello 'world'")   --> hello 'world'
```

Double hyphens (`'--'`) start a comment, which runs until the end of the line. Usually we use comments starting with `'-->'` to denote the result of some computation. (The symbol `'>'` after the double hyphen is already part of the comment.)

Lua supports another format for literal strings, called *long string*. A long string starts with two opening square brackets with any number of equal signs in between, and runs until two closing square brackets with the same number of equal signs in between:

```
print([["quotes" 'quotes']])   --> "quotes" 'quotes'
xml_code = [==[
<![CDATA[
 some data here
]]>
]==]
```

A long string may run for several lines and does not interpret escape sequences. This format is particularly useful for strings with code fragments, as we may insert the original fragment into Lua code without any changes.

The only data structure in Lua is the associative array, called *table* in Lua. Both the indices and values in a table may have any Lua value: numbers, strings, other tables, functions, etc. There is a special syntax for creating tables, called *constructor*. The constructor `{}` creates an empty table. The constructor `{x=1, y=2}` creates a table where index `"x"` (a string) has value 1 and index `"y"` has value 2:

```
t = {x=1, y=2}
print(t.y)     --> 2
```

The syntax `t.y` is sugar for `t["y"]`, that is, table `t` indexed by the string `"y"`.

A constructor like `{10,20,35}` creates a *list*, which is a table with numerical indices. In this example, index 1 has value 10, index 2 has value 20, and index 3 has value 35. We can also mix the formats: the constructor `{10,30,x=3}` creates a table where index 1 has value 10, index 2 has value 30, and index `"x"` has value 3:

```
t = {10, 30, x=3}
print(t[2], t.x)     --> 30   3
```

When called with multiple arguments, `print` prints each of them with tabs in between.

Functions in Lua may return multiple values. These multiple values may be used in a multiple assignment or passed straight to another function:

```
function foo (a, b) return a+b, a-b end
x, y = foo(20, 10)
print(x, y)          --> 30   10
print(foo(30, 4))    --> 34   26
```


DOI: 10.1002/spe

When the single argument to a function is a literal string or a table constructor, parentheses are optional:

```
print "hello"     --> hello
function printX (t) print(t.x) end
printX{x=10}      --> 10
```

Modules in Lua are simply tables:

```
require "math"
print(math.floor(3.14))   --> 3
```

In the first line, `require` is a predefined function, which is being called with a single literal string as argument. This function finds and runs the respective module definition, which builds a table with its exports and stores it in the global variable `math`. In the second line, the expression `math.floor` has the same meaning as `t.x`: 'the field indexed by the string `"floor"` in table `math`'.

Objects in Lua are also tables. Lua offers a special syntax for calling methods: `obj:foo(args)` is a shorthand for `obj.foo(obj, args)`. It gets the function stored at key `"foo"` in the object and calls it passing the object itself as the first argument. This first argument places the role of the *self* (or *this*). We can also implement objects using *userdata*. For Lua, a userdata is only a chunk of raw memory with an associated type. Any library written in C can create userdata values and use their raw memory to whatever it needs. Through overloading (see the following paragraph), a program may associate methods with userdata, which are called with the same colon syntax we use for calling methods in tables.

Lua also offers a mechanism to overload operators applied to tables and userdata. Each table or userdata has a table associated with it, which we call its *metatable*. When Lua evaluates an expression like `a+b`, where `a` or `b` is a table or a userdata, Lua looks into their metatable for a field with a reserved key `__add`. If present, Lua calls this field (which should be a function) to perform the operation. Otherwise Lua raises an error.

Metatables may provide most binary and unary operations and also indexing. When Lua tries to index a userdata, it looks in the userdata metatable for an `__index` field, and calls the associated function to perform the indexing. This is the mechanism that allows methods in userdata: the syntax `udata:foo(args)` translates to `udata.foo(udata, args)`; the indexing of `udata` triggers the `__index` function in its metatable; this function then returns a `foo` method, which is called by Lua.

3.2. The `re` module

The first module offered by LPEG, called `re` (from *regex*), creates patterns using strings written in a little language. In this text, when confusion may arise, we will enclose patterns for the `re` module inside marks like `|this|`. Inside Lua code, these strings are written like any other literal string in Lua; after all, for Lua they are just conventional strings.

We can use the `re` module like a conventional pattern-matching tool, as here:

```
require "re"
print(re.find("the number is 345", "[0-9]+"))            --> 15
print(re.find("the number is 345", "'number' / 'word'")) --> 5
```


DOI: 10.1002/spe

Function `re.find` returns the index of the first occurrence of a pattern in a subject, or nil if it cannot find a match.

The syntax for patterns in this little language is similar to the original PEG syntax that we described in the previous section, but with some differences, mainly to make it more appropriate for pattern matching:

- As we had already mentioned, the initial symbol is `pattern`, instead of `grammar`.
- As in Lua, literals may be enclosed both in single and double quotes.
- As in most other pattern-matching tools, character classes may start with a ^ to complement its meaning; for instance, `|[^()]|` matches any character that is not a parenthesis.
- There are some predefined patterns to match some character categories; for instance, `|%s|` matches any space character; `|%d|` matches a digit; `|%nl|` matches a newline character.
- Uses of non-terminal names must be enclosed in angle brackets.
- There is extra notation to express captures, which we will discuss later.

Another function provided by module `re` is `re.match`, which works in PEG style: it matches the pattern against a prefix of the subject and returns the index of the first character after the match:

```
print(re.match("words and words", "[a-z]+"))      --> 6
print(re.match("words and 3 numbers", "[0-9]*")) --> 1
```

As Lua indexes characters starting with 1, this numerical result is one more than the prefix length. If the match fails, `re.match` returns nil:

```
print(re.match("words and 3 numbers", "[0-9]+")) --> nil
```

We can also use grammars to describe patterns. The next example checks whether a string is a Lisp-style S expression:

```
P = [[
  ( S    <- <atom> / '(' %s* <S>* ')' %s*
    atom <- [a-zA-Z0-9]+ %s* ) !.
]]
s = "(a (b (x y) d ((x))) )"
print(re.match(s, P))      --> 24
s = "(a (b (x y) d ((x)) )"
print(re.match(s, P))      --> nil
```

Pattern `P` describes S-expressions: an S expression is either an atom or a sequence of S expressions enclosed in parentheses, with optional spaces (represented by `|%s*|`) between elements. Note how the entire grammar is parenthesized and concatenated with `|!.|`, to anchor it at the subject's end.

As a similar example, the next code parses arithmetic expressions:

```
P = [[
      Expression <- <Factor> ([+-] <Factor>)*
      Factor   <- <Term> ([*/] <Term>)*
      Term    <- <Number> / '(' <Expression> ')'
      Number   <- [0-9]+
    ]]
print(re.match("13+(22-15)", P))   --> 11
```

Later we will see how to add meaning to these grammars.

DOI: 10.1002/spe

3.3. The `lpeg` module

The second way to use LPEG is inspired by SNOBOL4 [15], where patterns are first-class values in the language. In this mode, we use standard Lua code to create patterns. Although verbose, this mode makes the entire Lua language available for writing patterns. The library offers several functions to create patterns and to combine them. It also uses the facilities of Lua for overloading operators to expose several of its functions as unary and binary operators. The next example illustrates the idea:

```
require "lpeg"

any = lpeg.P(1)             -- pattern that accepts one character
space = lpeg.S(" \t\n")     -- a set with the given chars
digit = lpeg.R("09")        -- a set with the range 0-9
lower = lpeg.R("az")        -- a set with the range a-z
upper = lpeg.R("AZ")        -- a set with the range A-Z

letter = lower + upper      -- '+' is an ordered choice
alnum = letter + digit

print(letter:match('a'))  --> 2
print(digit:match('u'))   --> nil
```

This piece of code is plain Lua code. Function `lpeg.P`, when given a number, creates a pattern that matches that number of characters; it is equivalent to the SNOBOL function `LEN`. Function `lpeg.S` creates a pattern that matches any character in the given string; it is equivalent to the SNOBOL function `ANY`. Module `lpeg` overloads the plus operator so that it denotes ordered choice when applied over patterns[**]. In the last two lines of the example we called the method `match` over two patterns.

All LPEG operators coerce several Lua types into patterns. Whenever a string is used where a pattern is expected, it is converted to a pattern that accepts only that string, literally. A boolean false gives a pattern that always fails, and true gives a pattern that always succeeds (the empty pattern). A number n is converted to a pattern that accepts n characters, like the result of `lpeg.P(`n`)`. Besides the omnipresent 1, which is equivalent to the dot in regex notation, other numbers are useful for matching against fixed-width fields and binary data.

The Lua exponentiation operator is overloaded to denote repetition. Unlike its usual meaning in formal languages, where p^n means n repetitions, in module `lpeg` `p^n` means *at least n* repetitions[††]. Specifically, `p^0` is equivalent to the PEG expression |p*| (zero or more repetitions), and `p^1` is equivalent to the PEG expression |p+| (one or more repetitions). We can create a typical pattern to match identifiers as follows:

```
identifier = (letter + "_") * (alnum + "_")^0
```

[**]We could have used the `'/'` operator to denote an ordered choice, as it does in PEGs, but its precedence is too high.

[††]There is no explicit primitive operation that corresponds to p^n, that is, exactly n repetitions. We can achieve this behavior by concatenating n copies of p.

DOI: 10.1002/spe

It reads as 'a letter or an underscore followed by zero or more alphanumeric characters or underscores.' The binary operator `*`, when applied over patterns, denotes concatenation[‡‡].

The *not* predicate is expressed by the unary minus operator. For instance, the following pattern succeeds only at the end of a string, where it cannot match any character:

```
EOS = -lpeg.P(1)     -- End Of String (equivalent to |!.|)
```

The binary minus operator also has a meaning. The construction `p1 - p2` is equivalent to the PEG expression `|!p2 p1|`, meaning 'matches an instance of `p1` as long as `p2` does not match.' For character sets, this corresponds to set difference:

```
nonLetter = 1 - letter
```

The *and* predicate is expressed by the `#` prefix operator. For instance, the following pattern matches the string `"for"` only if it is followed by a space:

```
Cident = "for" * #space
```

For simple patterns, module `re` is usually much more convenient, due to its terseness. For more complex patterns, however, the explicit construction supported by module `lpeg` allows a piecemeal definition style with several advantages: we can document each part better, we can test sub-patterns independently, we can reuse sub-patterns, and we can define auxiliary constructions through functions that create patterns. As most examples in this paper are small, I will favor the regex syntax here. In real use, most users prefer to use module `lpeg`.

The explicit constructors also simplify the implementation, because they do not need any parsing. Module `lpeg` is the basic module. The regex syntax, in turn, is implemented in LPEG itself, using explicit constructors [16].

The two modules offered by LPEG are not disconnected. The function `re.compile` compiles a regex string into a first-class pattern:

```
Bal = re.compile(" B <- '(' ([^()] / <B>)* ')' ")
print(Bal:match("(1 3 (4) ())"))   --> 13
P = identifier * space^0 * Bal
print(P:match("foo (a, b, c)"))    --> 14
```

Pattern `Bal` matches parenthesized expressions, and pattern `P` matches simple function calls. The grammar in the definition of `Bal` reads as 'a parenthesized expression is an open parenthesis, followed by a sequence of no-parenthesis characters or parenthesized expressions, followed by a closing parenthesis.'

3.4. Grammars

The SNOBOL-like modus operandi allows us to define patterns incrementally, with each new pattern using previously defined ones. Grammars pose a problem for this bottom-up approach, because we cannot build a recursive structure bottom up.

LPEG solves this problem using Lua tables. To create a grammar, we create a table where each entry represents a production rule. The call `lpeg.V(name)` builds a pattern that represents the non-terminal `name` in the grammar (`V` stands for *variable*). Of course, such a pattern makes sense

[‡‡]In formal languages it is common to denote concatenation like multiplication, with juxtaposition or a dot.

DOI: 10.1002/spe

only within a grammar, which still does not exist when `lpeg.V` is called. Hence, the result of that call is what we call an *open* pattern. Once the table is complete, we call the function `lpeg.P` to *close* it: this call creates a pattern to represent the complete grammar, linking open non-terminals to their corresponding definitions.

As an example, we (re)create below the grammar for balanced parenthesized expressions:

```
abs = lpeg.P{"B",  -- this is the initial symbol
  B = '(' * ((1 - lpeg.S('()')) + lpeg.V('B'))^0 * ')'
}
print(abs:match'((a b) c)')  --> 10
```

When the value for field `B` is evaluated, the table does not exist yet. The expression `lpeg.V('B')` creates an open reference to a non-terminal called `'B'`. When `lpeg.P` receives the complete table, it corrects this open reference to point to the field indexed by `'B'` in the table, therefore creating a recursive pattern.

As is the case for expressions, for such small grammars the regex syntax is easier to use. However, when grammars grow, and also when they include semantic actions (see below) written in Lua, many LPEG users prefer this SNOBOL-like syntax.

3.5. Searching

The main function in module `lpeg` is the `lpeg.match` function. Unlike most parsing tools, and like most pattern-matching tools, `lpeg.match` does not try to match the entire subject. It follows the PEG formalism, which dictates that a match does not need to consume the whole input. However, unlike most pattern-matching tools, and like most parsing tools, `lpeg.match` does not search for the pattern in the subject. Instead, it matches only 'anchored' in the beginning of the subject.

When we want to search for a pattern with LPEG, we write the search in the pattern itself. Given any pattern P, the pattern `|S <- P / . <S>|` either matches P or skips one character and tries again. In other words, it searches for the first occurrence of P in the subject. As we will see in Section 5, these search patterns are quite efficient.

We have already used function `re.find`. This function is just a wrapper that, given a pattern, creates the above grammar and matches it against the subject. It also uses a *position capture*, which we will discuss later, so that a successful match returns the position where the pattern was found.

An alternative way of expressing a search for the first occurrence of P in the subject is the pattern `|(!P .)* P|`: it skips as many characters as possible while not matching P, and then matches P.

Because the programmer writes the search loop, it is easy to adapt it to different needs. For instance, if we want to find a pattern P only at a word boundary, we can use a search pattern like this:

```
S <- P / [A-Za-z]* [^A-Za-z]+ <S>
```

It tries to match P; if it fails, it skips the current word plus the following non-word characters, and then repeats.

As another variation, let us consider searching for the *last* occurrence of a pattern in the subject. With traditional tools, where repetition is greedy but non-blind, we may search for the last occurrence of any pattern by prefixing it with `^.*`, which means 'go as far as possible and then match the

DOI: 10.1002/spe

pattern'. In LPEG this trick does not work, because repetition is blind. However, there are other tricks. A simple one is to repeat the search as many times as possible:

```
( S <- P / . <S> )*
```

The last successful search finds the last occurrence.

The previous trick only works if the last occurrence does not overlap other occurrences. If this is not the case, we may use something a little more complex, like this:

```
( S <- &P . / . <S> )*
```

This pattern repeatedly searches for P, but it uses the *and* predicate to check for the pattern without consuming it. Instead, after finding a new occurrence of P it consumes one single position (with the dot) and tries again.

```
p = "( S <- &'xuxu' . / . <S> )*"
print(re.match("xuxuxuxui", p))    --> 6
```

Because of the dot, the final result is one plus the position of the last occurrence.

3.6. Captures

As we mentioned in the Introduction, *captures* are an important facility of a pattern-matching tool. Captures in LPEG not only may capture information from the subject, but they may also combine and manipulate that information. From this point of view, the name 'capture' is misleading; the mechanism is actually a mix of conventional captures with semantic actions. (This description seems quite apt, since LPEG is actually a mix of pattern matching with parsing.)

Usually, the `match` function returns the index of the first character in the input string after the match. However, if the pattern makes any captures, `match` returns instead the values of these captures:

```
print(re.match("hello world", "{[a-z]+}"))  --> hello
```

A pattern enclosed in curly brackets captures the string that matches it[§§]; hence, the pattern `|{[A-Za-z]+}|` matches a word and captures it.

There is an important difference between captures in LPEG and in other pattern-matching tools. In most of these tools, if the pattern describes one capture, it will capture at most one value, even if the capture is inside repetitions. When there are multiple matches for the same capture, only the last one is preserved. In LPEG, by contrast, each match captures a new value. When there are multiple matches for the same capture, LPEG gets all of them. For instance, consider the next pattern, describing a list of words:

```
P = "( [^A-Za-z]* {[A-Za-z]+} )*"
```

Each match against this pattern captures a new value; a call to `re.match` will return all these values:

```
print(re.match("a few words", P))  --> a   few   words
```

[§§] Several regex tools use parentheses to mark captures; LPEG reserves parentheses for its traditional meaning of grouping.

DOI: 10.1002/spe

In this example, `re.match` returns three independent values, each the result of an independent capture.

Another simple capture is denoted by `|{}|`; this capture returns the position where the match (against an empty string) occurred. For instance, suppose we change the definition of 'word' in the previous example to `|{}[A-Za-z]+|`, that is, a position capture followed by one or more letters. Then we get the following output, where each number is the position of a new word:

```
print(re.match("a few more words", P))  --> 1   3   7   12
```

A *table capture* is denoted by the suffix operator `|->{}|`. It collects all captures done by a pattern into a list, implemented as a table in Lua. Let us add it to our previous example:

```
P = "( [^A-Za-z]* {}[A-Za-z]+ )* -> {}"
t = re.match("too many other words", P)
```

Now the result of the match, stored in variable `t`, will be the table `{1, 5, 10, 16}`, which is a word index into the subject: `t[i]` is the position of the `i`th word.

As another interesting example, let us return to our pattern for S expressions. Here we repeat that example, but adding captures to it:

```
S <- <atom> / '(' %s* <S>* -> {} ')' %s*
atom <- { [a-zA-Z0-9]+ } %s*
```

In the definition of `atom` we created a regular capture, by enclosing it in curly brackets. In the definition of `S` we created a table capture, by suffixing `|->{}|` to the expression `|<S>*|`. This operator may be read as 'send all captures from `<S>*` into a table', so that now that table is the capture. If we parse the string `"(a b (c d) ())"` against this new pattern, the result is the table `{"a","b",{"c","d"},{}}`, which exactly reflects the structure of the original S expression.

Another kind of capture allows us to create a new string with the captured values. The expression `|p -> string|` results in a copy of `string` where each mark `%n` is replaced by the *n*th capture from `p`. Consider the following example:

```
P = "({[0-9]^2} '-' {[0-9]^2} '-' {[0-9]^4}) -> '%3/%2/%1'"
print(re.match("16-09-1998", P))  --> 1998/09/16
```

The sub-pattern inside the parentheses specifies a date with two digits for days, two for months, and four for years. It also captures each of these numbers, by enclosing their respective sub-patterns in curly brackets. Then the outer capture (denoted by the suffix `'->'` operator) 'sends' these captures to the string `'%3/%2/%1'`. When the match occurs, `%1` is replaced by `'16'`, `%2` is replaced by `'09'`, and `%3` is replaced by `'1998'`.

LPEG offers several other kinds of captures. For instance, in a *function capture*, we 'send' the captures of a sub-pattern as arguments to a function so that the function result is the new captured value. We will not explore all options here, but we will describe one more capture, for substitutions.

3.7. Substitutions

Besides searching, one of the most common operations with pattern matching is substitution (called *replacement* in SNOBOL4).

Most pattern-matching tools treat searching and substitution as different operations. LPEG unifies these operations through a new form of capture, the *substitution capture*.

DOI: 10.1002/spe

In the regex syntax, we denote a substitution capture by enclosing a pattern with `{~...~}` (tilde curly brackets). A substitution pattern is somewhat similar to a basic capture; it captures the part of the subject that matches its enclosed pattern:

```
print(re.match("hello", "{~ . ~}"))      --> h
print(re.match("hello", "{~ .* ~}"))     --> hello
```

The difference happens when the enclosed pattern also has captures. In this case, for any capture nested inside the substitution match, the matched substring is replaced by its corresponding capture value:

```
P = "{~ ([A-Z] -> '+' / .)* ~}"
print(re.match("Hello", P))           --> +ello
print(re.match("Hello World", P))     --> +ello +orld
```

In this example, inside the substitution capture we have the capture `|[A-Z]->'+'|`. Hence, each match against `|[A-Z]|` is replaced by the value of its capture, `'+'`.

A substitution may enclose multiple captures to perform simultaneous substitutions:

```
P = "{~ ('0' -> '1' / '1' -> '0' / .)* ~}"
print(re.match("1101 0110", P))       --> 0010 1001
```

A substitution capture is a capture like any other. It does not need to comprise the entire pattern; it can be used inside other captures to avoid an extra step to correct or adapt captured values. The next example illustrates this usage. It shows a more advanced use of captures to parse comma-separated values (CSV), as described in RFC 4180 [17].

The following pattern is all we need to fully decode a CSV record:

```
record <- (<field> (',' <field>)*)->{} (%nl / !.)
field <- <escaped> / <nonescaped>
nonescaped <- { [^,"%nl]* }
escaped <- '"' { ([^"] / '""'->'"')* ~} '"'
```

Let us read that pattern. A record is a sequence of fields separated by commas, ending with an end of line or end of string (the pattern `|!.|`). The capture `->{}` collects in a table the captures from all fields.

A field can be escaped or non-escaped. A non-escaped field is a sequence of characters not including commas, double quotes, or newlines; its value is captured by a regular capture.

An escaped field is enclosed by double quotes. To write a double quote inside an escaped field, the quote must be escaped by preceding it with another double quote. To decode this encoding, we capture the value of an escaped field with a substitution capture and specify that two double quotes must be replaced by one (with the capture `|'""' -> '"'|`).

The final result of a match against that pattern is a list with the field values already unescaped. For instance, a match against the string

```
first,"second","hi, ho""hi"
```

will result in the list

```
{'first', 'second', 'hi, ho"hi'}
```


DOI: 10.1002/spe

4. THE PARSING MACHINE

Back in 1972, Aho and Ullman [9] presented an algorithm to parse any GTDPL (PEG ancestor's formalism) in linear time. The algorithm is based on the determinism of these grammars: given a non-terminal and a subject's position, either the non-terminal matches n characters or it fails. The algorithm builds a table that, for each pair $\langle nonterminal \times input\ position \rangle$, stores the respective result of the match. This table can be filled backwards, from the end of the subject to its beginning, in constant (for a given grammar) time for each entry.

A serious inefficiency of the above algorithm is that most values that it computes are never used; for each input position, there is generally only a small number of non-terminals that can be considered to match there. In 2002, Ford proposed *Packrat* [10], an adaptation of the original algorithm that uses lazy evaluation to avoid that inefficiency.

Even with this improvement, however, the space complexity of the algorithm is still linear on the subject size (with a somewhat big constant), even in the best case. As its author himself recognizes, this makes the algorithm not befitting for parsing 'large amounts of relatively flat' data [10, p. 57]. However, unlike parsing tools, regular-expression tools aim exactly at large amounts of relatively flat data.

To avoid these difficulties, we did not use the Packrat algorithm for LPEG. To implement LPEG we created a virtual parsing machine, not unlike Knuth's *parsing machine* [18], where each pattern is represented as a program for the machine. The program is somewhat similar to a recursive-descendent parser (with limited backtracking) for the pattern, but it uses an explicit stack instead of recursion.

Knuth's original parsing machine linked non-terminal calls with backtracking. A call to a non-terminal 'sub-routine' saved the current subject position; a sub-routine failure returned control to the callee and backtracked the subject to the saved position. Because of this link between calls and backtracking, that parsing machine could not express a generic grammar. Instead each rule should have the following form:

`S <-` A_1 `/` A_2 `/ ... / B C D`

That is, only the last option could have more than one non-terminal. To see why this restriction was necessary, consider the following rule:

```
S <- A B / C D
```

If `A` succeeded and then `B` failed, the current position would return to the position saved when `B` was called, not to the position saved when `S` was called. The only way to return to that position is by failing `S`. Only the last option can do that, because there are no other options to try.

To avoid this restriction, our machine has one instruction to push a backtrack option and another to call a non-terminal. This architecture not only allows unrestricted grammar rules but also allows a direct encoding for other constructs like repetitions and optionals, as we will see shortly.

The virtual machine keeps its state in four registers:

$$State = \langle \mathbf{N} \cup \mathbf{Fail}, \mathbf{N}, StackEntry^*, Capture^* \rangle$$

The registers have the following meanings:

1. The *current instruction* keeps the index of the next instruction to be executed (a natural number); it may also have a special **Fail** value, meaning that some match failed and the machine must backtrack.

DOI: 10.1002/spe

2. The *current subject position* keeps the current position in the subject (a natural number).
3. The *stack* is a list with two kinds of entries:

$$\mathit{StackEntry} = \mathbf{N} \cup (\mathbf{N}, \mathbf{N}, \mathit{Capture}^*)$$

The first kind represents return addresses (a natural number). Each non-terminal translates a call to its corresponding production; when that production finishes in success it must return to the point after the call, which will be at the top of the stack.

The second kind of entry represents pending alternatives (backtrack entries). Whenever there is a choice, the machine follows the first option and pushes on the stack information on how to pursue the other option if the first one fails. Each such entry comprises the instruction to follow in case of failure plus all information needed to backtrack to the current state (that is, the subject position and the capture list).

4. The *capture list* keeps information about captures made by the pattern:

$$\mathit{Capture} = (\mathbf{N}, \mathbf{N})$$

Each entry stores the subject position and the index of the instruction that created the entry, wherein there is extra information about the capture.

The machine has nine basic instructions. Figure 2 gives a formal description for these instructions, showing how each of them modifies the current state. In that description, $\mathcal{S}$ represents the subject (a sequence of characters). Each rule shows the state before executing the instruction, the current instruction, an optional condition, and the state after executing the instruction. The last two rules apply whenever the machine is in the **Fail** state, no matter the current instruction.

The following list gives a textual description for each instruction:

`Char` ***char***: Checks whether the character in the current subject position is equal to *char*. If it is equal, the machine consumes the current character and goes to the next instruction. Otherwise it fails. (See instruction `Fail` for details of what the machine does in this case.)

`Jump` ***label***: Jumps to instruction *label*. All instructions that need a label express the label as an offset from the current instruction, so that it is easy to relocate code. Jump instructions are used to organize grammars and to implement proper tail calls.

$$\begin{array}{rlcl}
\langle p,\, i,\, e,\, c\rangle & \texttt{Char}\ x,\ \mathcal{S}[i] = x & \Rightarrow & \langle p+1,\, i+1,\, e,\, c\rangle \\
\langle p,\, i,\, e,\, c\rangle & \texttt{Char}\ x,\ \mathcal{S}[i] \neq x & \Rightarrow & \langle \mathbf{Fail},\, i,\, e,\, c\rangle \\
\langle p,\, i,\, e,\, c\rangle & \texttt{Jump}\ l & \Rightarrow & \langle p+l,\, i,\, e,\, c\rangle \\
\langle p,\, i,\, e,\, c\rangle & \texttt{Choice}\ l & \Rightarrow & \langle p+1,\, i,\, (p+l, i, c) : e,\, c\rangle \\
\langle p,\, i,\, e,\, c\rangle & \texttt{Call}\ l & \Rightarrow & \langle p+l,\, i,\, (p+1) : e,\, c\rangle \\
\langle p_0,\, i,\, p_1 : e,\, c\rangle & \texttt{Return} & \Rightarrow & \langle p_1,\, i,\, e,\, c\rangle \\
\langle p,\, i,\, h : e,\, c\rangle & \texttt{Commit}\ l & \Rightarrow & \langle p+l,\, i,\, e,\, c\rangle \\
\langle p,\, i,\, e,\, c\rangle & \texttt{Capture}\ k & \Rightarrow & \langle p+1,\, i,\, e,\, (i, p) : c\rangle \\
\langle p,\, i,\, e,\, c\rangle & \texttt{Fail} & \Rightarrow & \langle \mathbf{Fail},\, i,\, e,\, c\rangle \\
\langle \mathbf{Fail},\, i,\, p : e,\, c\rangle & \mathit{any} & \Rightarrow & \langle \mathbf{Fail},\, i,\, e,\, c\rangle \\
\langle \mathbf{Fail},\, i_0,\, (p, i_1, c_1) : e,\, c_0\rangle & \mathit{any} & \Rightarrow & \langle p,\, i_1,\, e,\, c_1\rangle
\end{array}$$

Figure 2. Basic instructions for the parsing machine.

DOI: 10.1002/spe

`Choice` ***label***: Creates a backtrack entry in the stack, saving the current machine state plus the given label as the alternative instruction.

`Call` ***label***: Calls instruction *label*; that is, saves in the stack the address of the next instruction and jumps to instruction *label*. Call instructions implement non-terminals.

`Return`: Returns from a call, popping an instruction address from the stack and jumping to it. Because a complete pattern never leaves entries in the stack, there cannot be pending alternatives in the top of the stack when a rule returns.

`Commit` ***label***: Commits to a choice; this instruction simply discards the top entry from the stack and jumps to *label*. This top entry cannot be a return address, because a commit is always preceded by a choice.

`Capture` ***extra-info***: Adds an entry into the capture list with the current subject position and current instruction. If the complete pattern matches, a postprocessor traverses the capture list and, using the pointers to the instructions that created each entry, builds the capture values. (We will not discuss this postprocessor in this paper, as it does not participate in the matching algorithm.)

`Fail`: Forces a failure. First, the machine pops any return addresses from the top of the stack. Then, if the stack is empty, the machine halts and the whole pattern fails. Otherwise, the machine pops the top backtrack entry and assigns the saved values to their respective registers. With this assignment, the machine backtracks its current subject position and its capture list to the state they were when that backtrack entry was created, and jumps to the alternative instruction given in the `Choice` operator that created the entry.

`End`: The machine returns signalling that the match succeeded. This instruction appears only as the last one of a complete pattern.

Although this basic instruction set is enough to implement all the machine's functionality, it may result in long sequences of instructions for some common situations. For instance, we can check whether a character is a lower-case letter through a sequence of 26 choices. Of course, an explicit `Charset` instruction can be much more efficient. Because we want a practical and efficient machine, and not merely a theoretically complete one, we will add new instructions whenever we can improve performance. For a start, we add the following two instructions:

`Charset` ***set***: Checks whether the character in the current subject position belongs to the set *set*. Sets are represented as bit sets, with one bit for each possible value of a character. Each such instruction uses 256 extra bits, or 16 bytes, to represent its set[¶¶]. It handles success and failure like the `Char` instruction.

`Any` ***n***: Checks whether there are at least *n* characters in the current subject position. It handles success and failure like the `Char` instruction. This instruction could be replaced by a sequence of `Charset` instructions with all bits set, but an explicit instruction is much smaller and a little faster.

Figure 3 gives a formal description for these instructions and a few others that we will describe later.

Now, let us see how patterns translate to programs for the parsing machine. Here we will refer mainly to the `lpeg` module, because its operations are the primitive operations of LPEG. In the

[¶¶]Currently, LPEG does not have direct support for multibyte characters.

DOI: 10.1002/spe

$$
\begin{array}{rlcl}
\langle p,\, i,\, e,\, c\rangle & \texttt{Charset}\ X,\ \mathcal{S}[i] \in X & \Rightarrow & \langle p+1,\, i+1,\, e,\, c\rangle \\
\langle p,\, i,\, e,\, c\rangle & \texttt{Charset}\ X,\ \mathcal{S}[i] \notin X & \Rightarrow & \langle \textbf{Fail},\, i,\, e,\, c\rangle \\
\langle p,\, i,\, e,\, c\rangle & \texttt{Any}\ n,\ i+n \leq |\mathcal{S}| & \Rightarrow & \langle p+1,\, i+n,\, e,\, c\rangle \\
\langle p,\, i,\, e,\, c\rangle & \texttt{Any}\ n,\ i+n > |\mathcal{S}| & \Rightarrow & \langle \textbf{Fail},\, i,\, e,\, c\rangle \\
\langle p_0,\, i_0,\, (p_1, i_1, c_1) : e,\, c_0\rangle & \texttt{PartialCommit}\ l & \Rightarrow & \langle p_0 + l,\, i_0,\, (p_1, i_0, c_0) : e,\, c_0\rangle \\
\langle p,\, i,\, e,\, c\rangle & \texttt{Span}\ X,\ \mathcal{S}[i] \in X & \Rightarrow & \langle p,\, i+1,\, e,\, c\rangle \\
\langle p,\, i,\, e,\, c\rangle & \texttt{Span}\ X,\ \mathcal{S}[i] \notin X & \Rightarrow & \langle p+1,\, i,\, e,\, c\rangle \\
\langle p,\, i,\, h : e,\, c\rangle & \texttt{FailTwice} & \Rightarrow & \langle \textbf{Fail},\, i,\, e,\, c\rangle \\
\langle p_0,\, i_0,\, (p_1, i_1, c_1) : e,\, c_0\rangle & \texttt{BackCommit}\ l & \Rightarrow & \langle p_0 + l,\, i_1,\, e,\, c_1\rangle
\end{array}
$$

Figure 3. Extra instructions for the parsing machine.

following manipulations, we assume that all patterns always end with an implicit `End` instruction, but this last instruction is always discarded when the pattern is used to compose other patterns.

4.1. Code for basic constructions

A literal string translates to a sequence of `Char` instructions, one for each character in the string. For instance, the call `lpeg.P("ana")` returns the following program:

```
Char 'a'
Char 'n'
Char 'a'
```

The call `lpeg.P(`*n*`)` translates to an instruction `Any` *n*. Both sets and ranges translate to a `Charset` instruction.

The program for a concatenation `p1 * p2` is the concatenation of the programs for `p1` and `p2`. The operation receives `<p1>` and `<p2>` (the programs that represent patterns `p1` and `p2`) and returns the following program:

```
<p1>
<p2>
```

4.2. Code for ordered choice

The ordered choice `p1 + p2` translates to the following program:

```
    Choice L1
    <p1>
    Commit L2
L1: <p2>
L2: ...
```

The program starts saving the machine state (`Choice` instruction) and then runs `<p1>`. If `<p1>` succeeds, running to completion, the program executes `Commit L2`, which removes the saved state from the stack and jumps to the end of the pattern (label `L2`). If `<p1>` fails, the machine backtracks to the initial saved state and jumps to `L1`, to try `<p2>`. If `<p2>` also fails then the whole choice fails, because it has no other alternative in the stack.

The ordered-choice construction does a trivial optimization when both `p1` and `p2` are character sets. In this case, `p1 + p2` translates to a single `Charset` instruction with the union of both sets.

DOI: 10.1002/spe

A more elaborate optimization for choices concerns associativity. Although the PEG semantics for alternatives is associative, its coding in the parsing machine is not. If we follow the original construction, the code for `(p1 + p2) + p3` would be like this:

```
    Choice L1
    Choice L2
    <p1>
    Commit L3
L2: <p2>
L3: Commit L4
L1: <p3>
L4: ...
```

However, the code for `p1 + (p2 + p3)` would be this:

```
    Choice L1
    <p1>
    Commit L2
L1: Choice L3
    <p2>
    Commit L2
L3: <p3>
L2: ...
```

It is not difficult to see that the second code is faster than the first when `<p1>` succeeds: the first code executes two `Choice` operations plus two commits, while the second code executes only one of each. The more the alternatives we have, the worse the left associativity becomes when compared with right associativity: the code for *n* alternatives would demand *n* choices plus *n* commits when the first alternative succeeds.

Addition in Lua is left associative (as in most languages); hence, a plain `p1 + p2 + p3` would generate bad code. To avoid this, the compilation algorithm for ordered choices checks whether the first pattern is already a choice and, if needed, reconstructs the code into the right-associative form.

4.3. Code for repetition

The repetition construction `p^0` could generate the following program:

```
L1: Choice L2
    <p>
    Commit L1
L2: ...
```

This program saves the machine state and runs `<p>`. If `<p>` fails, the program backtracks and ends (jumping to label `L2`). Otherwise, the program commits to the new state and repeats.

In a typical loop, this program executes repeatedly the sequence `Commit L1; L1: Choice L2`. We can optimize this sequence in two ways. First, we create a new instruction, called `PartialCommit`, to represent the sequence. Second, this new instruction does not actually need to perform a commit followed by a choice. Instead of removing an entry from the stack and adding a new one, the instruction simply updates the top entry. The alternative label (`L2`) remains the

DOI: 10.1002/spe

same; hence, `PartialCommit` updates only the current subject position and the capture list (see Figure 3). With this instruction, the program for a loop looks like this:

```
     Choice L2
L1:  <p>
     PartialCommit L1
L2:  ...
```

LPEG provides another optimization for the case where the pattern being repeated is a character class. This is a very common case, arising from patterns like `|[A-Za-z]+|` and `|[ \t\n]*|`. These loops have a dedicated instruction, `Span` *charset*, that consumes a maximum span of input characters belonging to the given character set. This maximum span may have zero length; hence, this instruction never fails.

4.4. Code for predicates

The *not* predicate, denoted in Lua as `-p`, could be translated like this:

```
     Choice L2
     <p>
     Commit L1
L1:  Fail
L2:  ...
```

This program starts by saving the current state. It then proceeds to execute `<p>`. If this fails, the machine backtracks to the saved state and goes to label `L2`, therefore continuing normal execution. If, however, `<p>` succeeds, the following `Commit` removes the backtrack option, so that the `Fail` after it fails the entire pattern, as desired. Again we can optimize this construction with a new instruction, `FailTwice`. As the name implies, `FailTwice` behaves like two consecutive fails: it removes the top entry from the stack (which must be the pending alternative pushed by the `Choice` instruction) and then fails. With this new instruction, the *not* predicate is coded like here:

```
     Choice L1
     <p>
     FailTwice
L1:  ...
```

The difference operator for patterns, denoted by `p1 - p2`, is usually coded following its definition: `-p2 * p1` (where `-p2` is coded as we just showed). The case when both `p1` and `p2` are character sets, however, deserves a special treatment. In this case, their difference is coded as a single charset instruction with the set difference between the two patterns.

The *and* predicate could be coded following its definition as a double *not* predicate. However, even with the original instruction set we could do a little better than that, coding `&p` like here:

```
     Choice L1
     Choice L2
     <p>
L2:  Commit L3
L3:  Fail
L1:  ...
```


DOI: 10.1002/spe

If <p> does not fail, the `Commit` instruction will remove the top backtrack entry (pushed by the second choice instruction), so that the following `Fail` will 'fail' to `L1`, backtracking to the initial position and continuing the match. If <p> fails, the control will also go to `L2`, but this time consuming the top backtrack entry. The commit will consume the next backtrack entry (pushed by the first choice) and the `Fail` instruction will make the pattern fails, as there are no other backtracks left from this pattern.

Once more we create a new instruction to optimize a particular construction. The new instruction is `BackCommit`, which behaves like a mix of a fail and a commit: Like a `Fail`, `BackCommit` backtracks to the state stored in the stack; like a `Commit`, it jumps to the given label after popping the stack entry (instead of jumping to the backtrack label). With `BackCommit`, we can code the *and* predicate as follows:

```
     Choice L1
     <p>
     BackCommit L2
L1:  Fail
L2:  ...
```

If <p> does not fail, the `BackCommit` instruction backtracks to the initial subject position and jumps to the pattern's end. If <p> fails, the control goes to `L1`, wherein the whole pattern fails.

4.5. Code for grammars

The opcode for grammars is mostly straightforward: each non-terminal translates to a `Call` opcode, and each rule ends with a `Return` opcode.

When we create a non-terminal, it is still not part of a grammar. Hence, LPEG uses a special instruction, `OpenCall`, to represent that pseudo-pattern. For instance, the code for `'(' * lpeg.V("B") * ')'` is like this:

```
Char '('
OpenCall "B"
Char ')'
```

When LPEG receives the complete grammar (as a Lua table), then it acts like a link editor. It creates a new pattern with the concatenation of all the rules in the grammar, each one ending with a `Return` instruction, and then traverses the result correcting each `OpenCall` instruction into a `Call` to the appropriate offset[|||]. Because the result must be a pattern that matches the first rule, this link editor adds at the beginning of the pattern a call to its first rule followed by a jump to its end.

As an example, consider the following LPEG grammar:

```
g = lpeg.P{"S", -- start symbol
    S = lpeg.V"B" + (1 - lpeg.S"()"),    -- S <- <B> / [^()]
    B = '(' * lpeg.V"S" * ')'             -- B <- '(' <S> ')'
}
```

[|||] Like a link editor, it gives an error if there is a reference to a non-existent rule.

DOI: 10.1002/spe

The resulting code is as follows:

```
    Call S
    Jump L2

 S: Choice L4
    Call B
    Commit L5
L4: Charset [^()]
L5: Return

 B: Char '('
    Call S
    Char ')'
    Return

L2: End
```

It may seem that we could put the `End` instruction after the initial `Call`, without the jump. However, an important characteristic of LPEG is that grammars result in regular patterns. We can get the result of a grammar and use it like any other pattern, even inside other grammars. To allow this composability, grammars follow the rule that a pattern ends with its only `End` instruction.

A particularly important optimization for grammars regards tail calls. This optimization is done by the link editor. When it sees a `OpenCall` instruction followed by a `Return` instruction, it translates the `OpenCall` to a `Jump`, instead of a `Call`. Hence, for instance, the usual idiom for searching a pattern, such as `|X <- 'ana' / . <X>|`, translates to a regular loop:

```
    call X
    Jump L2
 X: Choice L1
    Char 'a'
    Char 'n'
    Char 'a'
    Commit L3
L1: Any 1
    Jump X
L3: Return
L2: ...
```

As with other forms of tail calls, more important than the performance gain is the fact that successive calls do not accumulate on the stack.

LPEG uses a conservative algorithm to check for and reject patterns with infinite loops. The algorithm does a symbolic execution of the pattern, checking whether it could loop without consuming any input. The algorithm is not exact, as this problem is undecidable [8]. Instead, it conservatively assumes that any predicate may always succeed, without consuming any input. For instance, it rejects the pattern `|(&'a' &'b')*|`, even though this pattern cannot loop (because the loop body

DOI: 10.1002/spe

$$
\begin{array}{llcl}
\langle p,\, i,\, e,\, c\rangle & \texttt{TestChar}\ x\ l,\ \mathcal{S}[i] = x & \Rightarrow & \langle p+1,\, i+1,\, e,\, c\rangle \\
\langle p,\, i,\, e,\, c\rangle & \texttt{TestChar}\ x\ l,\ \mathcal{S}[i] \neq x & \Rightarrow & \langle p+l,\, i,\, e,\, c\rangle \\
\langle p,\, i,\, e,\, c\rangle & \texttt{TestCharset}\ X\ l,\ \mathcal{S}[i] \in X & \Rightarrow & \langle p+1,\, i+1,\, e,\, c\rangle \\
\langle p,\, i,\, e,\, c\rangle & \texttt{TestCharset}\ X\ l,\ \mathcal{S}[i] \notin X & \Rightarrow & \langle p+l,\, i,\, e,\, c\rangle \\
\langle p,\, i,\, e,\, c\rangle & \texttt{TestAny}\ n\ l,\ i+n \leq |\mathcal{S}| & \Rightarrow & \langle p+1,\, i+n,\, e,\, c\rangle \\
\langle p,\, i,\, e,\, c\rangle & \texttt{TestAny}\ n\ l,\ i+n > |\mathcal{S}| & \Rightarrow & \langle p+l,\, i,\, e,\, c\rangle \\
\langle p,\, i,\, e,\, c\rangle & \texttt{Choice}\ l\ d & \Rightarrow & \langle p+1,\, i,\, (p+l, i-d, c) : e,\, c\rangle
\end{array}
$$

Figure 4. Instructions for head-fail optimizations.

always fails, given that `|&'a'|` and `|&'b'|` cannot both succeed for the same input). In practice, even these rare false positives are benign, as they usually signal some problem in the pattern (e.g. it can never succeed).

4.6. Optimizations based on *head fails*

A *head fail* occurs when a pattern fails at its very first check. For many patterns, head fails are much more frequent than non-head fails. For instance, if we search for the word 'ana' in a typical English text, more than 90% of all fails will be head fails[***]. When searching for a balanced parenthesized expression in a program source, practically 100% of all fails will be head fails against the initial '('.

Without optimizations, a head fail is somewhat costly. Typically, it involves a `Choice` operator followed by a failing check operator (`Char` or `Charset`). Both operations are expensive, when compared with other operations: the choice must save the entire machine's state, and the failing check must restore that state. Given the cost and the frequency of head fails, it seems worthwhile to optimize them.

For this optimization, we introduce three new instructions (one for each check instruction):

`TestChar` ***char label***: Checks whether the character in the current subject position is equal to *char*. If it is equal, the machine consumes the current character and goes to the next instruction. Otherwise it jumps to *label*.

`TestCharset` ***set label***: Checks whether the character in the current subject position belongs to the set *set*. It handles success and failure like the `TestChar` instruction.

`TestAny` ***n label***: Checks whether there are *n* characters in the current subject position. It handles success and failure like the `TestChar` instruction.

Besides these new instructions, we need also to modify the `Choice` instruction, adding an *offset* to it. Now, when this instruction saves the current machine state, it saves the current subject position minus this offset. Figure 4 gives a formal description for these new instructions and for the modified `Choice` instruction.

Let us see an example to understand how these changes work. Suppose we have the following pattern, which searches for the word 'ana' in a string:

```
A <- 'ana' / . <A>
```

[***]Assuming a frequency of around 8% for the letter 'a.'

DOI: 10.1002/spe

When we compile it, we get the following loop:

```
L1: Choice L2
    Char 'a'
    Char 'n'
    Char 'a'
    Commit L3
L2: Any 1
    Jump L1
L3: End
```

This loop spends most of its time doing the sequence `Choice L2`, `Char 'a'` (which fails most of the time, backtracking to `L2`), `Any 1`, and then back to the start with `Jump L1`. With the new instructions, we translate that loop as follows:

```
L1: TestChar 'a' L2
    Choice L2 1
    Char 'n'
    Char 'a'
    Commit L3
L2: Any 1
    Jump L1
L3: End
```

Now, the critical sequence becomes `TestChar 'a' L2` (which fails, jumping to `L2`), `Any 1`, and `Jump L1`. Not only the new sequence is one (expensive) instruction shorter than the old one, but it also replaces a somewhat expensive (when failing) `Char` instruction by a cheaper `TestChar`.

When that first `TestChar` succeeds, it consumes the current character (`'a'`). However, if the pattern subsequently fails, it should backtrack the subject position to where it was before reading that `'a'`. To provide for that, the `Choice` instruction in the pattern has its offset set to 1, so that the subject position it saves is one less than the current position—that is, it saves the position where the match began.

Once we have these test instructions, several other optimizations are possible. An important one applies when the acceptable first characters of each pattern in an ordered choice are disjoint. For instance, consider the following pattern for a string with balanced parentheses:

```
B <- ( '(' <B> ')' / [^()] )*
```

The code for the ordered choice inside the repetition will be like this:

```
L1: TestChar '(' L2
    Call L1
    Char ')'
    Jump L3
L2: Charset [^()]
L3: ...
```

The first option in the ordered choice can only start with an open parenthesis, whereas the second option cannot start with an open parenthesis. Therefore, if the first `TestChar` succeeds, we know

DOI: 10.1002/spe

the second option cannot succeed. There is no need for a `Choice` instruction at all, because the given alternative would always fail.

4.7. Finite automata

As we saw in Section 2, it is very easy to translate a finite automata into a PEG. A nice property of our optimizations is that, when we translate a deterministic automata into a PEG, the resulting grammar generates quite efficient code. For a certain kind of language, the resulting code behaves exactly like the original automata.

The translation of a generic finite automata into a grammar generates rules like this:

A <- a_1 <A_1> / ⋯ / a_n <A_n>

When the automata is deterministic, all a_i in each rule must be distinct. In this case, the optimizations we saw previously eliminate the `Choice` instructions, so that the code for each choice becomes like this:

```
    TestChar a_i
    Call A_i
    Jump L
    ...
L: Return
```

However, now the call becomes a tail call; hence, it is optimized to a jump:

```
TestChar a_i
Jump A_i
...
```

As this happens to all options in all rules, the parsing machine behaves exactly like a state machine: at each state (rule), it tests some conditions and then jumps to the next state.

Final states present an interesting situation. Usually, the fact that a state is an acceptance state is translated as an extra rule for that state reducing to the empty string. If that state has transitions that may lead to later acceptance (that is, if the defined language does not have the prefix property), then the resulting code for that state cannot be fully optimized. To see why, consider a rule like this:

```
A <- 'a' <B> / ''
```

If `B` fails, the machine will succeed with the second option; hence, it must push a backtrack entry before calling `B`.

If the language described by the automaton has the prefix property (that is, no string in the language has proper substrings in the language), then final states have only one option (the empty string) and the resulting code runs like a state machine.

4.8. Code for captures

The basic code for captures is straightforward. A capture simply adds two `Capture` instructions in the pattern, one at its beginning and the other at its end. Hence, for instance, the code for `|{'ana'}|`

DOI: 10.1002/spe

would be like this:

```
Capture begin
Char 'a'
Char 'n'
Char 'a'
Capture end
```

Both `begin` and `end` are attributes with no effect on the matching. As we saw already, each `Capture` instruction saves an entry on the capture list with the current subject position plus a pointer to the instruction itself. After the match, this list plus the subject string are enough to build the capture values.

This obvious coding has two bad effects on the machine. First, the initial `Capture` adds a wasted overhead in the frequent case when the pattern fails. Second, and worst, it hinders other optimizations. For instance, the `Char` instruction following it cannot be replaced by a `TestChar` instruction by other optimizations, because the `TestChar` would not undo the capture when failing.

A simple way to avoid these pitfalls is to move the `Capture` instruction as late in the instruction sequence as possible. To be able to do this, we add to the instruction a correction offset: an instruction `Capture begin 3` will push an entry pointing to three positions in the subject before the current one. With this offset, the code for `|{'ana'}|` can be like this:

```
Char 'a'
Char 'n'
Char 'a'
Capture begin 3
Capture end
```

Now, the capture instructions are executed only when there is a match, and the initial `Char` instruction can be changed by other optimizations.

Sometimes, the move of the beginning capture instruction puts it just before its corresponding closing instruction, as it happened in the previous example. In this case, we can collapse the two instructions into a single one:

```
Char 'a'
Char 'n'
Char 'a'
Capture full 3
```

5. PERFORMANCE

In this section we study the performance of the parsing machine. We start developing an analytical model and then we perform some benchmarks against other pattern-matching tools. We also analyze the effectiveness of some optimizations that we described in the previous section and finally provide some information about code size.

DOI: 10.1002/spe

5.1. Analytical model

The parsing machine, like most backtracking machines, has a worst-case time complexity that is exponential in the subject size. Restricted backtracking reduces the number of patterns with such behavior, but does not eliminate them. For a simple example, consider the following pattern:

```
A <- ..<A> / .<A> / '@'
```

It is easy to see that the sequence A_n, the number of steps to match `A` against an input of size n, satisfies the equation $A_n = A_{n-2} + A_{n-1} + 1$ and therefore has exponential growth.

This worst-case behavior, however, is no impediment to the use of the parsing machine. First, it does not happen often in practical patterns. Second, and more importantly, we can predict, understand, and eventually correct this behavior, because the parsing machine gives us a clear performance model for reasoning.

As we already saw, the parsing machine may parse any regular language in linear time, by using a right-linear grammar. (If the regular language has the prefix property then the parse uses constant space.) This result, however, is not very useful in practice, because a right-linear grammar seldom reflects the way we need to parse the language. (In other words, we cannot add the captures we need.) Hence, let us have a look at more generic patterns.

The time T to match a character (or any, or a character set) is constant:

$$T(\texttt{'x'}) = T(\texttt{.}) = T(\texttt{[}\cdots\texttt{]}) = O(1)$$

The time to match an ordered choice is the sum of the times for each option (as the first one may fail):

$$T(\texttt{p1 / p2}) = T(\texttt{p1}) + T(\texttt{p2})$$

The worst time to match a sequence is also the sum of the times for each component, because each component may be a predicate and therefore consume no input:

$$T(\texttt{p1 p2}) = T(\texttt{p1}) + T(\texttt{p2})$$

Predicates and optionals have the same time of the original pattern:

$$T(\texttt{!p}) = T(\texttt{\&p}) = T(\texttt{p?}) = T(\texttt{p})$$

A repetition pattern must consume at least one character at each iteration—otherwise, it would cause an infinite loop and would be rejected by the compiler. Hence, it may repeat at most n times, where n is the input length. On the other hand, each iteration may traverse the entire remaining input and yet consume only one character, due to failures or predicates (which are a kind of failure). That implies that each iteration may take the worst-case time of the repeating pattern. Hence, we have that

$$T(\texttt{p*}) = nT(\texttt{p})$$

That gives us the following result:

The worst-case time complexity to match a string of length n against a fixed pattern with no grammatical rules is $O(n^k)$, where k is the pattern's star height (that is, the maximum nesting of repetitions in the pattern).

DOI: 10.1002/spe

The proof is a simple induction over the pattern's structure.

For non-recursive rules in a grammar, we can apply the previous equations, simply expanding each rule with its definition. For recursive rules, we can express their time complexity by a recurrence relation, as we did for the first example in this section. More specifically, suppose that we have a recursive definition like `A <- ... <A> ...` . This pattern must consume at least one character before recursively calling `A` again—otherwise the grammar would be left recursive and rejected by the compiler. Hence, if the input size for the first call is n, we may assume that this size becomes at most $n-1$ for the recursive call. We then may express the time $T(\texttt{A})_n$ as a function of $T(\texttt{A})_{n-1}$; solving the equation gives us the worst-case complexity for the rule.

In practice, even quadratic time complexity may be too slow for pattern matching. Hence, the previous worst-case analysis is not very useful, because many interesting patterns have nested repetitions (and therefore a star height of at least 2). Hopefully, for many of these patterns a more careful analysis can establish a tighter upper bound. As a simple example, consider the following pattern, which matches a list of words separated by spaces:

```
([A-Za-z]+ [ \t\n]*)*
```

Although its star height is 2, it is easy to establish that its behavior is linear with the input, as it fails at most once for each input character (except at the match's end). With future work we hope to establish tighter upper bounds for common constructions with nested repetitions and recursive grammars.

With one important exception, the *ad hoc* optimizations in the parsing machine do not change the complexity of matches; hence, they do not affect the previous discussion. The important exception is the tail call optimization, which reduces linear (stack) space to constant space. Although the basic case is easy to model, some occurrences of tail calls result from other optimizations, and therefore are not easily modelled. For instance, an analysis of a deterministic right-linear grammar would not indicate that it could work with constant space. We still have to work out a better space model for this situation.

5.2. Benchmarks

Let us see now how LPEG compares with other pattern-matching implementations. Before we proceed, we should note that we will not be comparing apples with apples. On the one hand, LPEG has more expressive power than other pattern-matching tools, as it can express a superset of the deterministic context-free languages. On the other hand, other tools have several features currently lacking in LPEG, such as Unicode support.

Despite these caveats, we think the comparison is worthwhile. We should not look at the results as hard benchmarks, drawing conclusions like 'this implementation is faster than that.' Nevertheless, we can draw conclusions about the appropriateness of LPEG as a pattern-matching tool, to be used in the same kind of tasks programmers use other pattern-matching tools.

We will compare LPEG with both POSIX regex and PCRE. I chose these two tools because they are well known and widely used, and also because they have a readily available interface to Lua (Lrexlib 2.2). For completeness, we will also measure the standard Lua pattern-matching library. This is a very simple tool, implemented in less than 500 lines of C code, that supports only patterns without grouping or alternatives.

For all tests we used LPEG version 0.8, Lua 5.1, PCRE version 6.7 04 July 2006, and the POSIX regex functions from the GNU C Library stable release version 2.5. The tests ran on a Linux

DOI: 10.1002/spe

Table I. Time (milliseconds) for searching a string in the Bible.

Pattern	PCRE	POSIX regex	Lua	LPEG	LPEG(2)	False starts
`'@the'`	5.3	14	3.6	40	9.9	0
`'Omega'`	6.0	14	3.8	40	10	8853
`'Alpha '`	6.7	15	4.2	40	11	17851
`'amethysts'`	27	38	24	47	21	256897
`'heith'`	32	44	26	50	23	278986
`'eartt'`	40	53	36	52	26	407883

machine (Ubuntu 7.04) with a Pentium 4 2.93 GHz and 1.5 GB of RAM. Following Kernighan's lead [19], we used as subject for all tests the full text of the King James Bible[†††]. The whole text has 4 432 995 ASCII characters distributed in 99 830 lines.

The first group of tests search for literal patterns in the subject. The literals are either not present in the text or appear only near its end. For LPEG, we transform a pattern *p* into a search with the following function, which builds the grammar |`S <-` *p* `/ . <S>`|:

```
function search (p)
 p = re.compile(p)  -- compile pattern as a regex
 return lpeg.P{ p + any * lpeg.V(1) }
end
```

Moreover, we must quote all literals for LPEG, but not for the other tools. Table I shows the results. (For now, ignore the column LPEG(2).) From the results we see that the search for the initial character dominates the time: the more frequent the first character (and therefore, the more frequent the false starts), the slower the search. This dependency occurs in all tools, but it is weaker in LPEG than in the other tools.

In the traditional pattern-matching tools the search loop is built in, whereas in LPEG we use an explicit loop (the tail-recursive grammar) for searching. With few false starts, the search loop dominates the total time, and hence traditional tools benefit from their built-in search. As the number of false starts increases, the actual matching algorithm becomes increasingly relevant to the total search time. As we see in the table, as the matching algorithm becomes more relevant, the edge of PCRE/POSIX over LPEG decreases. In the last entry, LPEG is as fast as POSIX, despite its explicit loop.

As we have just seen, the explicit search loop in LPEG slows down the search, because the loop must be interpreted by the parsing machine instead of being compiled into the tool code. However, exactly because the loop is explicitly written by us, we may try to optimize it. We learned that the search for the first character in the pattern dominates the loop time; hence, we could try to optimize this search. Instead of trying to match the pattern at every subject's position, we may try it only when the first character is correct. Assuming that *x* is the first character of pattern *p*, the

[†††] We used the file `http://www.gutenberg.org/dirs/etext90/kjv10.txt` from the Project Gutenberg, with the header (289 lines) removed.

DOI: 10.1002/spe

next grammar does the trick:

```
S <- p / . [^x]* <S>
```

First it tries p. If that fails, it escapes one position, then loops until the next x and repeats. The next function builds this search pattern automatically for a given string `p`:

```
local any = lpeg.P(1)
function search (p)
 local x = string.sub(p, 1, 1)   -- first character from 'p'
 return lpeg.P{ p + any * (any - x)^0 * lpeg.V(1) }
end
```

With this search loop we get results 2–4 times faster than with the original loop, as presented in column 'LPEG(2)' of Table I.

Table II shows the times for some more complex searches. Unlike our previous cases, all these patterns have no fixed prefix. The first pattern matches any word longer than 13 letters. (The first such word in the Bible is 'Jegarsahadutha', at line 2834.) The second pattern matches any word followed by 'Abram'. (It finds 'begat Abram', at line 847.) The last pattern is similar, but for 'Joseph' instead of 'Abram'. ('name Joseph' appears later in the text, at line 2612). For all these patterns, LPEG has the fastest times.

Table III shows some results when searching for the last occurrence of a pattern in the subject. As we explained in Section 3.5, we can do this search in traditional tools by prefixing the pattern with `^.*`, which expands as far as it can. (In PCRE we also add the prefix `(?s)` so that the dot matches newlines, too.) In LPEG we suffix the searching pattern with a `*`, so that the search repeats as many times as possible.

For some patterns, PCRE finds the last occurrence of the pattern more than five times faster than it finds the first occurrence. However, if the pattern does not appear in the subject (e.g. `@the`) or its last appearance is far from the end (e.g. `Tubalcain`), the same search becomes three orders of magnitude slower, because PCRE backtracks all its way from the subject's end. For some more

Table II. Time (milliseconds) for searching a pattern in the Bible.

Pattern	PCRE	POSIX regex	Lua	LPEG
`[a-zA-Z]{14,}`	10	15	16	4.0
`([a-zA-Z]+) *'Abram'`	16	12	12	1.9
`([a-zA-Z]+) *'Joseph'`	51	30	36	5.6

Table III. Time (milliseconds) for searching the last occurrence of a pattern in the Bible.

Pattern	PCRE	POSIX regex	Lua	LPEG
`'Omega'`	0.1	58	23	40
`'@the'`	252	58	120	40
`'Tubalcain'`	110	56	119	39
`([a-zA-Z]+) *'Abram'`	*Error*	670	766	258
`[a-zA-Z]{14,}`	8.2	1446	30	141

DOI: 10.1002/spe

Table IV. Time (milliseconds) for matching a pattern with the Bible.

Pattern	Narwhal	LPEG
`S <- 'Omega' / . <S>`	*Error*	35
`(!'Omega' .)* 'Omega'`	116	44
`S <- (!<P> .)* <P> ; P <- 'Omega'`	$> 10^6$	98

complex patterns (like any word followed by `Abram`), PCRE does not even complete the search, giving a `PCRE_ERROR_MATCHLIMIT` error. For another somewhat similar pattern (words with at least 14 letters), PCRE again finds its last occurrence faster than its first.

Because LPEG is based on PEGs, we could compare it with other PEG tools, too. When comparing LPEG with other PEG tools, it is important to keep in mind that they have different goals. Current PEG tools aim at traditional parsing, not pattern matching. The foremost difference against LPEG is that other tools are compiled, not interpreted. Like Yacc, these tools translate the grammar into a piece of code in some target language that is then compiled into a parser. LPEG (and other pattern-matching tools) treats the grammar/pattern dynamically, during program execution.

Another difference, which we already mentioned, is that several PEG implementations use some form of memorization to avoid backtracking. Memorization is very handy when you have complex grammars and do not want to adjust them to avoid backtracking. A typical example is the if–then–else construct. In PEG, we can write this construct as follows:

```
if <- 'IF' space exp 'THEN' space stat 'ELSE' space stat
    / 'IF' space exp 'THEN' space stat
```

With LPEG, every *if* statement without an *else* part would be parsed twice, once for the first option (which fails when trying to match `'ELSE'`) and again for the second option.

Usually, it is not difficult to modify the grammar to avoid the backtracking, like here:

```
if <- 'IF' space exp 'THEN' space stat ('ELSE' space stat)?
```

However, for a complex grammar, these modifications may be tedious.

We already argued that memorization may not be appropriate for the large files and flat grammars that are common in pattern matching. To substantiate this claim, we did some tests with the Narwhal Compiler Suite [20]. This tool translates a PEG into C++ code, which is then compiled into a parser. We chose it because of the performance of its target language (C++). However, even compiled C++ code does not compensate for the overhead of memorization, as we can see in Table IV. In the first test, the parser overflows its stack, because it does not do proper tail calls. (This is not relevant for conventional parsing.) The second test does not involve any non-terminal symbol; hence, there is no memorization. Nevertheless, Narwhal is 2.6 times slower than LPEG. The third test repeats the second, moving a fixed pattern (`'Omega'`) into an independent rule. This small change forces Narwhal to memorize its results; we interrupted the test after 30 min.

5.3. Effectiveness of optimizations

When we described the parsing machine, we also described several optimizations for the machine. Here we analyze the effectiveness of those optimizations on some real patterns.

DOI: 10.1002/spe

The settings for this experiment is the same as from the previous benchmarks: the same machine (Pentium 4), the same operating system, and the same subject (the Bible).

We used the following patterns for this experiment:

1. `|S <- 'transparent' / . <S>|`—Searches for the word 'transparent' using tail recursion.
2. `|(!'transparent' .)* 'transparent'|`—Searches for the word 'transparent' using an explicit loop.
3. `|S <- 'transparent' / . [^t]* <S>|`—Again searches for the word 'transparent', using an optimization to skip false starts.
4. `|S <- [a-zA-Z]+ ' '* 'transparent' / . <S>|`—Searches for the first word before the word 'transparent' using tail recursion.
5. `|(!([a-zA-Z]+ ' '* 'transparent') .)*|`—Searches for the first word before the word 'transparent' using an explicit loop.

Note that, unlike the previous benchmarks, here we are not using an implicit search for each pattern; all searches are explicit. These explicit descriptions allow us to better understand how each optimization affects the pattern.

Figure 5 presents the results. Each group of columns shows the times to match each of the previous patterns. For each group we used a different version of LPEG 0.8, each with different optimizations disabled. All times are relative to a base case, not shown in the graph, that has all optimizations disabled. From this base case, each new version cumulatively enables some optimizations, as described next:

Partial Commit: Adds the `PartialCommit` instruction for loops.
Span: Adds the `Span` instruction for loops over character sets.
Fail Twice: Adds the `FailTwice` instruction to optimize the *not* predicate.
Head Fail: Adds the `Test*` instructions for the optimizations based on head fails.

As we see in the graph, The `PartialCommit` instruction speeds up all patterns that use explicit loops, reducing the match time by around 10%. Only the first pattern is unaffected, because it has no explicit loop.

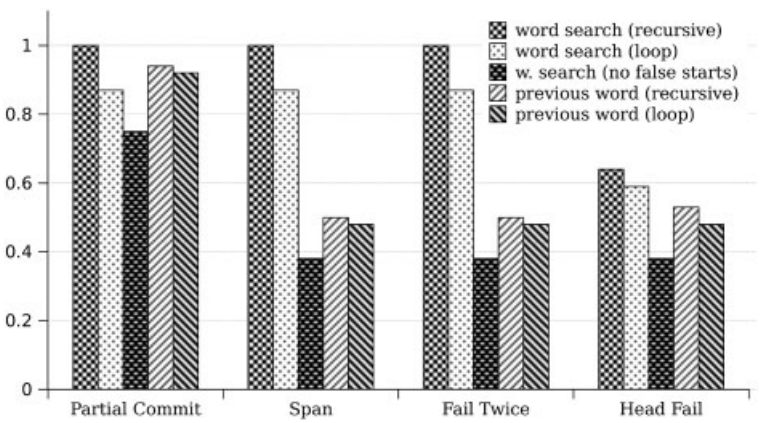

Figure 5. Times to match some patterns using different optimizations (relative to non-optimized matches).

DOI: 10.1002/spe

The `Span` instruction reduces the match time by almost 50% in the three last patterns, but of course it does not affect patterns 1 and 2, which do not have repetitions of character sets.

Two patterns in this test use the *not* predicate, but the `FailTwice` instruction has no perceptible effect in any of them. A probable explanation is that the instruction only executes when the *not* predicate fails, that is, when its body succeeds. In search patterns, this success happens at most once, when the search ends. Maybe other kinds of patterns may benefit from this instruction. Anyway, we think it is worth keeping this optimization, as it is very simple and reduces the size of the resulting code.

The last optimization in the graph adds the instructions to optimize head fails. It affects only the first two patterns, which have a high percentage of head fails. For these patterns, however, the speed up is palpable: it reduces the match time by 36% for the first pattern and by 32% for the second.

Overall, all optimizations combined reduce the average match time for these patterns by 48%.

5.4. Code size

As we had already mentioned, one of the attractiveness of the parsing machine is its simplicity. Here we provide some numbers about its implementation. These numbers refer to LPEG version 0.8.

As we saw, LPEG comprises two modules: `lpeg` and `re`. Module `lpeg` is implemented in 2200 lines of ANSI C code. The main chunks in this code are as follows:

- 220 lines of house-keeping code (type declarations plus some macros);
- 270 lines for the parsing machine *per se*;
- 140 lines for the infinite-loop checker;
- 900 lines for the functions that create patterns (the 'compiler'); of these lines, approximately 280 implement the optimizations;
- 340 lines for the post-processing of captures.

Module `re`, which implements the 'regular-expression' syntax for LPEG, is written in 200 lines of Lua code. It is so small because it uses LPEG itself (through module `lpeg`) to parse regular expressions. It implements a grammar somewhat similar to the one we used to describe PEGs in this paper, augmented with semantic actions to build a corresponding pattern as it matches an expression.

Because building patterns is expensive, module `re` memorizes previous compilations. For instance, the first time a program calls `re.search(s, "[a-z]+")`, the function compiles the pattern `|[a-z]+|`, uses the result to build the search pattern, and memorizes this final pattern. On subsequent searchings for `|[a-z]+|`, `re.search` reuses the memorized pattern.

For comparison, here are the code sizes of the other pattern-matching tools that we used in our benchmarks, PCRE and POSIX regexes. Of course, the same caveats from the benchmarks are valid here too.

The main modules in PCRE are the compiler, with 6221 lines of C; a pattern machine, with 4940 lines of C; and a `study` module, with 579 lines of C. This `study` module optimizes repeated matches against a fixed subject by pre-building some tables about the subject.

The main modules in POSIX regex are the compiler, with 3800 lines of C, and the pattern machine, with 4329 lines of C.

DOI: 10.1002/spe

6. CONCLUSION

We have implemented a pattern-matching tool based on Parsing Expression Grammars PEGs. The resulting library, LPEG, offers the expressive power of PEGs with the ease of use of regular expressions.

Besides being a practical option for common pattern-matching applications, LPEG seems specially suited for languages that are too complex for traditional pattern-matching tools but do not need a complex Yacc-lex implementation. Examples include domain-specific languages such as SQL and regular expressions, and even XML. For instance, LuaTex [21] is an extended version of pdfTeX that embeds Lua as its scripting language. It uses LPEG for tasks ranging from URL splitting to XML parsing. SciTE-tools, a variant of the SciTE text editor, uses LPEG for syntax highlighting.

LPEG presents some nice features and some novelties:

- It bases its patterns on a small set of primitives with formal semantics (PEGs), instead of regular expressions augmented with a myriad of *ad hoc* features. Its patterns can express most regex constructions (e.g. greedy, lazy, and possessive repetitions; anchors at the beginning and end of the subject; positive and negative lookaheads) plus complete grammars.
- It unifies in a single operation matching (parsing), searching, and substitutions. The unification of parsing with searching is achieved through an efficient implementation, so that search loops can be written as patterns. The unification of searching and substitutions is achieved with a new form of capture.
- It unifies captures and semantic actions. A pattern may return small independent pieces of the subject (like regex captures), entire abstract syntax trees, or anything in between.
- It uses a new parsing machine that is simple and efficient. Its performance is on par with other pattern-matching tools like PCRE. Its whole implementation comprises 2200 lines of C plus 200 lines of Lua. The parsing machine has a simple formal description based on operational semantics.

Although LPEG is written for Lua, it could be implemented easily for any scripting language. Nothing in LPEG is specific to Lua, except for captures, which are built on top of multiple returns and tables. This part of LPEG would have to be adapted to the new target language. Support for operator overloading is optional, as its use is mainly a syntactical convenience. Nevertheless, LPEG fits nicely into Lua philosophy: a small but powerful set of mechanisms instead of a bag of *ad hoc* features, plus a small and efficient implementation.

The implementation described in this paper is available under the MIT license at the following url:

`http://www.inf.puc-rio.br/~roberto/lpeg/`

ACKNOWLEDGEMENTS

I would like to thank Mike Pall, for useful comments about initial versions of LPEG; Luiz Henrique de Figueiredo, Gavin Wraith, and Noemi Rodriguez, for useful comments about earlier versions of this paper; Francisco Sant'anna, for helping with some benchmarks; Fabio Mascarenhas and Sergio Medeiros, for helping with the formal description of the parsing machine; and the anonymous reviewers, for several suggestions to improve this paper. This study was partially supported by the Brazilian Research Council (CNPq, Grant #300993/2005-6).

REFERENCES

1. Thompson K. Programming techniques: Regular expression search algorithm. *Communications of the ACM* 1968; **11**(6):419–422.
2. Cox R. Regular expression matching can be simple and fast. http://swtch.com/~rsc/regexp/regexp1.html [20 June 2008].
3. Laurikari V. NFAs with tagged transitions, their conversion to deterministic automata and application to regular expressions. *Seventh International Symposium on String Processing and Information Retrieval* (*SPIRE'00*), September 2000. IEEE: New York, 2000; 181–187.
4. Laurikari V. TRE matching library. http://laurikari.net/tre/ [20 June 2008].
5. Fowler G. An interpretation of the POSIX regex standard. http://www.research.att.com/~gsf/testregex/re-interpretation.html [20 June 2008].
6. Clarke CLA, Cormack GV. On the use of regular expressions for searching text. *ACM TOPLAS* 1997; **19**(3):413–426.
7. Wall L, Christiansen T, Orwant J. *Programming Perl* (3rd edn). O'Reilly Media: Sebastopol, CA, U.S.A., 2000.
8. Ford B. Parsing expression grammars: A recognition-based syntactic foundation. *Thirty-first Symposium on Principles of Programming Languages* (*POPL'04*), January 2004. ACM: New York, 2004.
9. Aho A, Ullman J. Limited backtrack parsing algorithms. *The Theory of Parsing*, *Translation and Compiling*, *Volume I*: *Parsing*, Chapter 6. Prentice-Hall: Englewood Cliffs, NJ, 1972.
10. Ford B. Packrat parsing: A practical linear-time algorithm with backtracking. *Master's Thesis*, Department of Electrical Engineering and Computer Science, MIT, September 2002.
11. Abigail. Reduction of 3-CNF-SAT to Perl regular expression matching. http://perl.plover.com/NPC/NPC-3SAT.html [20 June 2008].
12. Hutton G. Higher-order functions for parsing. *Journal of Functional Programming* 1992; **2**(3):323–343.
13. Ierusalimschy R, de Figueiredo Luiz H, Celes W. *Lua 5.1 Reference Manual*. Lua.Org, Rio de Janeiro, Brazil, 2006; ISBN 85-903798-3-3.
14. Ierusalimschy R. *Programming in Lua* (2nd edn). Lua.Org, Rio de Janeiro, Brazil, 2006; ISBN 85-903798-2-5.
15. Griswold RE, Poage JF, Polonsky IP. *The SNOBOL4 Programming Language* (2nd edn). Prentice-Hall: Englewood Cliffs, NJ, 1971.
16. Ierusalimschy R. LPeg—Parsing expression grammars for Lua. http://www.inf.puc-rio.br/~roberto/lpeg/lpeg.html [20 June 2008].
17. Shafranovich Y. Common format and MIME type for comma-separated values (CSV) files. *RFC 4180* (Informational), October 2005.
18. Knuth D. Top-down syntax analysis. *Acta Informatica* 1971; **1**:79–110.
19. Kernighan BW, Van Wyk CJ. Timing trials, or, the trials of timing: Experiments with scripting and user-interface languages. http://cm.bell-labs.com/cm/cs/who/bwk/interps/pap.html [20 June 2008].
20. Tisher G. The Narwhal compiler suite. http://sourceforge.net/projects/narwhal/ [20 June 2008].
21. Hagen H. The Luafication of TEX and ConTEXt. *TUGboat* 2008; **29**(2):295–302.

DOI: 10.1002/spe