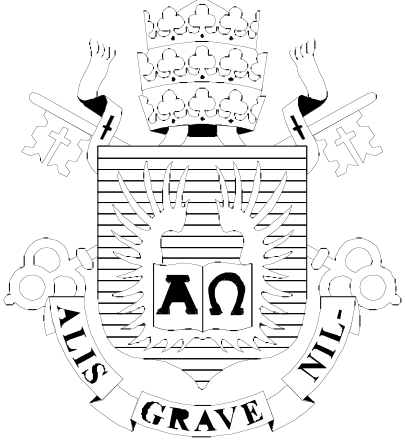

PUC

ISSN 0103-9741

Monografias em Ciência da Computação
n°08/08

A Text Pattern-Matching Tool based on Parsing Expression Grammars

Roberto Ierusalimschy

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO

RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22451-900

RIO DE JANEIRO - BRASIL

Monografias em Ciência da Computação, No. 08/08 ISSN: 0103-9741
Editor: Prof. Carlos José Pereira de Lucena February, 2008

A Text Pattern-Matching Tool based on Parsing Expression Grammars

Roberto Ierusalimschy

roberto@inf.puc-rio.br

Abstract:

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 (e.g., C identifiers) or cannot be described by regular expressions (e.g., parenthesized expressions). Moreover, the inherently non-determinism of regular expressions does not fit the need to capture specific parts of a match.

Moved 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 implementation.

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.

Keywords: Pattern matching, Parsing Expression Grammars, formal languages

Resumo:

Ferramentas atuais para casamento de padrão em texto se baseiam em expressões regulares. Entretanto, expressões regulares puras tem se mostrado um formalismo muito fraco para a tarefa: muitos padrões interessantes são ou difíceis de se escrever (e.g., identificadores C) ou não podem ser descritos por expressões regulares (e.g., expressões parentizadas). Além disso, o não-determinismo inerente a expressões regulares não combina bem com a necessidade de se capturar partes específicas de um casamento.

Por esses motivos, a maioria das linguagens de script atuais usa ferramentas de casamento de padrão que estendem o formalismo original de expressões regulares com um conjunto ad hoc de construções, como repetições gulosas, repetições preguiçosas ("lazy"), "regra do casamento mais longo", etc. Essas construções ad hoc trazem seus próprios problemas, como falta de fundamentos formais e implementação complexa.

Neste artigo, propomos o uso de PEGs ("Parsing Expression Grammars") como uma base para casamento de padrões. Seguindo esta proposta, apresentamos LPEG, uma

ferramenta de casamento de padrões baseada em PEGs para a linguagem de script Lua. LPEG unifica a facilidade de uso de ferramentas de casamento de padrões com todo o poder expressivo de PEGs. Devido a esse poder expressivo, LPEG evita as diversas construções ad hoc presentes em outras ferramentas para casamento de padrões. Também apresentamos uma Máquina de Análise Sintática ("Parsing Machine") que permite uma implementação compacta e eficiente de PEGs para casamento de padrões.

Palavras-chave: Casamento de padrões, PEGs, linguagens formais

In charge for publications:
Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC-Rio Departamento de Informática
Rua Marquês de São Vicente, 225 - Gávea
22451-900 Rio de Janeiro RJ, Brazil
Tel. +55 21 3527-1516 Fax: +55 21 3527-1530
E-mail: `bib-di@inf.puc-rio.br`
Web site: `http://bib-di.inf.puc-rio.br/techreports/`

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 [Tho68]. 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 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 not very difficult, but trickier than it seems) and C identifiers (which is very hard to define, because it should exclude reserved words like `for` and `int`).

Another, and stronger, limitation of the original theory of regular expressions is that it concerns only the recognition of a string, with no regard to 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 [Lau00, Lau], 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 [Fow].

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 [CC97], any search algorithm using the longest-match rule may need to traverse the subject multiple times.

Moved 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, PCRE (*Perl Compatible Regular Expressions*, 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, greedy and possessive repetitions, look ahead, look behind, non-backtracking subpatterns, etc. [WCO00]

- 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, and so they cannot ensure linear time complexity. Actually, several innocent-looking patterns require non-linear time. Moreover, these implementations have no performance models, so it is hard to predict the performance of any particular pattern. The following example, adapted from the Perl book [WCO00], runs in a few milliseconds in a typical workstation:

```
$_ = "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaac";
if (/a*a*a*a*a*a*b/) {
  printf "yes\n";
}
```

If we simply change the `b` in the pattern to `[b]` (which has the same meaning), the code runs almost 10,000 slower.

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

In this paper, we present a new pattern-matching tool based on *Parsing Expression Grammars* (PEGs) [For04]. 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 [AU72]. Although a PEG looks suspiciously similar to a Context-Free Grammar, it does not define a language, but rather an algorithm to recognize a language.

One of the main advantages of PEGs is that they can express a language syntax down to the character level, without the need for a separate lexical analysis. 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 [For04].
- PEGs allow a fast and simple implementation, based on a *parsing machine* (see Section 4).
- Although it is possible to write patterns with exponential time complexity for the parsing machine[1], they are much less common than in regexes, thanks to the limited backtracking. In particular, patterns written without grammatical rules have always 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 turn it into a 3-SAT solver.[2]

In the next section we will discuss PEGs in more detail. In Section 3 we present LPEG, a 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.

[1] Actually, there is an algorithm that matches any PEG in linear time with the subject size [AU72], but it also needs linear space, so we do not use it in LPEG. In Section 4 we will discuss these implementation aspects.

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

2 Parsing Expression Grammars

Parsing Expression Grammars (PEGs) are a formal system for language recognition based on *Top-Down Parsing Languages* (TDPL). [For04]

At first glance, PEGs look quite similar to Context-Free Grammars (CFGs) extended with some regular-expression operators. For instance, the following PEG recognizes arithmetic expressions:

```
Expression <- Factor ([+-] Factor)*
Factor <- Term ([*/] Term)*
Term <- Number / '(' Expression ')'
Number <- [0-9]+
```

As in regular expressions, `[+-]` means a plus or a minus sign, `a*` means zero or more `a`s, and `[0-9]+` means one or more digits. The slash (`/`) is the *ordered choice* operator, somewhat similar to the vertical bar (`|`) used in BNF. (PEGs also offer the conventional `?` posfix operator to denote an optional part.)

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 [AU72].

The ordered choice operator (in a choice `A / B`, `B` is only tried if `A` fails) makes the result of any parsing unambiguous. Given a PEG and an input string, either the parsing fails or it accepts a prefix of the input string in a unique way.

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:

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

When trying to match the input string against `A`, the parser will first try 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 a *global* backtracking).

This different 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 (such as "the longest match rule"). Repetitions in PEG may be greedy or non greedy, and blind or non blind.

A blind greedy repetition (sometimes called *possessive*) always matches the maximum possible span (therefore greedy), disregarding what comes next (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]*0` against `count1`, it is useless to try a shorter match for the `[a-z]*` part after the `1` fails against `0`. Nevertheless, this is what several backtracking tools do. (This behavior is the cause of the exponential time for the failure of a pattern like `a*a*a*a*[b]`, which we saw in the previous section.)

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

```
E <- e E / ''
```

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). So, 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 `.*b`, the matcher must find the *last* occurrence of `b` in the subject. Of course, to know that the occurrence is the last one the matcher must go until the subject's end (independently of the algorithm). If you need non-blind greedy repetition, you can write it with a PEG like this:

```
E <- e E / b
```

This pattern will always tries first the option `e E`, which consumes one instance of `e` and tries `E` again. This will go on until the subject's end. Once there, both `e` and `b` fails; it cannot match an `E` anymore and so the previous `e E` fails, too. The matcher then tries to match `b` at that previous position. If that also fails, it backtracks again, and again tries to match `b`. So, it backtracks step by step until it finds a `b` (or until there is no more backtracking and the entire pattern fails).

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 `a*?` for a non-greedy repetition), but PEGs do not need any special mechanism for it. If you want to match the minimum number of `e`s up to the first `b`, you can write the following:

```
E <- b / e E
```

This pattern first tries to match `b`. If that fails, it matches `e` and repeats. As we will see shortly, however, there is a more natural way to express non-blind non-greedy repetitions with PEGs.

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`.

Syntactic predicates greatly expand the expressiveness of PEGs, particularly because the not predicate provides a kind of a complement operator. For instance, to match strings that do not contain `"acac"` as a substring, the following pattern may be used: `(!'acac' .)* !.` It may be read as "zero or more occurrences of any character as long as the current position does not match `'acac'`, followed by the end of the subject." This is a typical example of a tricky pattern to express with strict regular expressions.

The not predicate also 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 `e` as long as `b` does not match, and then matches `b`".

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

```
E <- E X / ...
```

the result is an infinite loop. We say that such grammars are not *well formed* [For04]. Similarly, loops of patterns that can match the empty string (e.g., `''*`) are not well formed, as their expansion results in a left-recursive rule. Although the general problem of whether

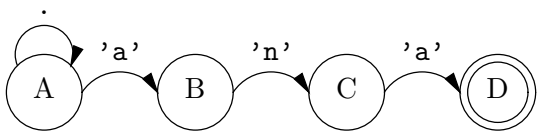

Figure 1: A NDFA that accepts strings ending with `'ana'`

a grammar is well formed is undecidable [For04], some PEG tools implement conservative algorithms to avoid these degenerate loops.

PEGs have a interesting relationship with right-linear grammars. A *right-linear grammar* is a grammar in which each production body has at most one non terminal, at its end. Because they have 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 NDFA 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 (e.g., strings with an even number of 0s and an odd number of 1s). An automatic translation from the automata to a regular expression usually results in an unusable expression. Using PEGs, we can translate the automata automatically to a usable grammar.

3 The LPEG Library

The LPEG library is an implementation of Parsing Expression Grammars for the Lua language [IdFC96, Ier06]. Unlike other PEG implementations, which aim at parsing, LPEG aims at pattern matching. Therefore, it offers a modus operandi somewhat similar to other pattern-matching tools, based on regular-expression syntax (but without compromising the full power of PEGs when needed). LPEG also offers a complementary interface, following the SNOBOL tradition, where patterns are first-class values built within the language.

In the first modus operandi, patterns are described by strings. The syntax for these strings is based on PEG syntax, but with some influence from regexes. As in PEGs, a string may describe a grammar but, differently from PEGs, it may also describe a single expression. For instance, the pattern `[A-Za-z]+` matches words. The pattern `[0-9]{2} '/' [0-9]{2} '/' [0-9]{4}` matches dates (with exactly two digits for days and months and four digits for the year); note that literals must be enclosed in quotes, and spaces outside quotes have no significance. The pattern `B <- '(' ([^()] / B)* ')'` matches balanced parenthesized expressions. (In words, a balanced expression is an opening parenthesis, followed by zero or more of either a non-parenthesis character or a balanced expression, followed by a closing parenthesis.) This string describes a grammar, and the resulting pattern matches its first (and, in this particular example, only) rule.

The second way to use LPEG is inspired by SNOBOL4 [GPP71], where patterns are values in the language. The library offers several functions to create patterns and to combine them. (Using Lua's metamethods, LPEG exposes several of these functions as unary and binary operators.) Although verbose, this option makes the entire Lua language available for writing patterns. The next example illustrates the idea:

```
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
```

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`). The plus operator denotes ordered choice.[3]

The exponentiation operator denotes repetition. Unlike its usual meaning in formal languages, where p^n means n repetitions, in LPEG `p^n` means *at least* n repetitions. Specifically, `p^0` represents the PEG expression `p*` (zero or more repetitions), and `p^1` represents the PEG expression `p+` (one or more repetitions). A typical pattern to match identifiers can be expressed as follows:

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

It reads as "a letter or an underscore followed by zero or more alphanumeric characters or underscores." The `*` operator denotes pattern concatenation. (In formal languages it is common to denote concatenation like multiplication, with juxtaposition or a dot.)

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)
```

The binary minus operator also has a meaning. The LPEG construction `p1 - p2` is equivalent to the PEG expression `!p2 p1`. 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 = lpeg.P("for") * #space
```

For simple patterns, the regex syntax is usually much more convenient, due to its terseness. However, for more complex patterns, the explicit construction 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.)

The explicit constructors also simplify the implementation, because they do not need any parsing. The regex syntax, in turn, is implemented in LPEG itself, on top of explicit constructors, in a total of less than 120 lines of Lua code. [Ier]

The two ways to use LPEG are not disconnected. The function `re.compile` compiles a regex string into a pattern, which then can be used like any other pattern. The next example, although artificial, shows this point:

[3] We could have used the division operator to denote an ordered choice, as it does in PEGs, but its priority is too high.

```
identifier = re.compile("[a-zA-Z]+")
spaces = lpeg.S(" \t\n")^0   -- equivalent to compile("[ \t\n]*")
Bal = re.compile(" B <- '(' ([^()] / B)* ')' ")
P = identifier * spaces * Bal^1
```

The final pattern `P` matches any identifier followed by spaces followed by one or more parenthesized expressions.

Grammars

Like for expressions, there are two ways to write grammars. The first one is using the regex-like syntax. For instance, the following code creates a pattern that matches the shortest non-empty prefix of a string of `a`'s and `b`'s that has the same number of `a`'s and `b`'s:

```
abs = re.compile[[
  S <- 'a' B / 'b' A
  A <- 'a' / 'b' A A
  B <- 'b' / 'a' B B
]]
```

(The notation `[[...]]` in Lua delimits literal strings with multiple lines.)

A grammar creates a pattern that matches its first rule. Once compiled, the resulting pattern may be used like any other pattern. For instance, we can use that previous pattern to build a new pattern that matches complete strings that have the same number of `a`'s and `b`'s:

```
P = abs^0 * -1
```

(In words, P matches a sequence of zero or more non-empty minimal segments with the same number of `a`'s and `b`'s followed by the end of the string.)

We can also create grammars in LPEG following the SNOBOL-like way. This second way to express grammars uses Lua tables. Each entry in the table represents a production rule. The call `lpeg.V(n)` builds a pattern that represents the nonterminal n in a grammar (`V` stands for *variable*). Of course, such pattern makes sense only within a grammar, which still does not exist when `lpeg.V` is called. So, the result of that call is actually an *open* pattern. The function `lpeg.P` accepts a table containing a whole grammar with open patterns and fixes them, returning a pattern that represents a given non-terminal of the grammar.

As an example, we repeat below the previous grammar for strings of `a`'s and `b`'s in this other syntax:

```
abs = lpeg.P{"S",   -- this is the initial symbol
  -- S are minimal strings with same number of 'a's and 'b's
  S = 'a' * lpeg.V"B" + 'b' * lpeg.V"A",
  -- A are minimal strings with one extra 'a'
  A = 'a' + 'b' * lpeg.V"A" * lpeg.V"A",
  -- B are minimal strings with one extra 'b'
  B = 'b' + 'a' * lpeg.V"B" * lpeg.V"B"
}
```

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.

Matching

LPEG offers a single function for matching, called `match`. It receives a pattern and a string, and tries to match the pattern against a prefix of the string. By default, `match` returns the index of the first character in the string after the matching prefix,[4] or the special value `nil` if the match fails:

```
print(re.match("Hello", "[A-Z]*"))    --> 2
print(re.match("hello", "[A-Z]*"))    --> 1
print(re.match("hello", "[A-Z]+"))    --> nil
```

Unlike most parsing tools, and like most pattern-matching tools, LPEG 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. (If needed, the pattern `"!."` may be used to anchor the pattern at the subject's end.) However, unlike most pattern-matching tools, and like most parsing tools, LPEG does not search for the pattern in the subject. Instead, it matches only "anchored" in the beginning of the subject. This restriction may seem like a severe limitation in the usefulness of LPEG as a tool for pattern matching, because searching is one of the main uses of these tools. But there is a good reason for this restriction.

LPEG does not search for the pattern in the subject because it is very easy to specify the search in the pattern itself. Given any pattern `P`, the pattern `(!P .)* P` skips as many characters as possible while not matching `P`, and then matches `P`. In other words, it matches the first occurrence of `P` in the subject.

Because patterns are first-class values, it is easy to write a function that, given any pattern `P`, builds a pattern that searches for `P`:

```
local any = lpeg.P(1)
function search (p)
  return (any - p)^0 * p
end
```

Of course, we can write directly the search function:

```
function search (subject, pattern)
  pattern = (lpeg.P(1) - pattern)^0 * pattern
  return pattern:match(subject)
end
```

Another option for searching `P` is the pattern `S <- P / . S`: it either matches `P` or skips one character and tries again. As we will see in Section 5, these search patterns are quite efficient.

Because the programmer writes the search loop, it is easy to adapt it for special needs. For instance, if we want to find a pattern `P` only at a word boundary, we can build the 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 (`[A-Za-z]*`) plus the following non-word characters (`[^A-Za-z]+`), 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 pattern". In LPEG this trick does not work,

[4] As Lua counts characters starting with 1, this result is one more than the prefix length.

because repetition is blind. But, given the function `search` we just defined, the pattern `search(p)^0` does the trick. It simply repeats the search as many times as possible: the last successful search finds the last occurrence.

Summing up, in most pattern-matching tools the default is to search for the pattern, but we may anchor the match using an ad hoc facility (the `^` assertion). In LPEG, the default is an anchored match, but we may do the search using the standard mechanisms of PEGs.

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 union of conventional captures with semantic actions. (This description seems quite apt, since LPEG is actually a union of pattern matching with parsing.)

Usually, the `match` function returns (one plus) the size of the prefix of the subject that matched the given pattern. However, if the pattern makes any captures, `match` returns instead the values of these captures. For instance, consider the next pattern, describing a list of words:

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

A pattern enclosed in curly brackets captures the string that matches it,[5] so the pattern `{[A-Za-z]+}` matches a word and captures it. Each match against this pattern captures a new value; a call to `re.match` will return all these values:[6]

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

Note the subtle difference from captures 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.

Another simple capture is denoted by `{}`; this capture returns the position where the match (against an empty string) occurred. For instance, if 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), we get the following output:

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

Each number is the position where a new word begins.

The *table capture*, denoted by the suffix operator `->{}`, collects all captures done by a pattern into a list (implemented as a table in Lua). As an interesting example, consider the next definition for Lisp-style S expressions:

```
S <- atom space / '(' space S* ')' space
atom <- [a-zA-Z0-9]+
space <- [\n\t ]*
```

(In words, an S expression is either an atom or a sequence of S expressions enclosed in parentheses, with optional spaces between elements.) Now we add captures to it:

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

[6] A function in Lua may return multiple values.

```
S <- atom space / '(' space S* -> {} ')' space
atom <- { [a-zA-Z0-9]+ }
space <- [\n\t ]*
```

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 has been 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 subpattern 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 subpatterns 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, we can "send" the captures of a subpattern as arguments to a function so that the function result is the new captured value. I will not explore all options here, but I will describe one more capture, for substitutions.

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*.

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]->'+'`. So, 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 [Sha05].

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

```
record <- (field (',' field)*)->{} ('\n' / !.)
field <- escaped / nonescaped
nonescaped <- { [^,"\n]* }
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, and 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 (`'""' -> '"'`).

The final result of a match against it 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'}
```

4 The Parsing Machine

Back in 1972, Aho & Ullman [AU72] presented an algorithm to parse any Generalized Top-Down Parsing Language (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 $\langle \textit{non terminal} \times \textit{input position} \rangle$, 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* [For02], which is 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 ([For02], p. 57). However, unlike parsing tools, regular-expression tools aim exactly at large amounts of relatively flat data.

To avoid these difficulties, LPEG is not based on the Packrat algorithm. The implementation of LPEG is based on a virtual parsing machine, not totally unlike Knuth's *parsing machine* [Knu71], 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 <- A1 / A2 / ... / 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 was failing `S`. Only the last option can do that, as 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:

The current subject position keeps the current position in the subject.

The current instruction keeps the index of the next instruction to be executed.

The stack keeps track of return addresses. Each non terminal translates to 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 stack keeps also 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 all information needed to backtrack to the current state (that is, the subject position and the capture list length) plus the instruction to follow in case of fail.

The capture list list keeps information about captures made by the pattern. It plays no role in the matching process. Each entry stores the subject position and a pointer to the instruction that created the entry (wherein there is extra information about the capture).

The machine has the following nine basic instructions:

`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.)

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

`Fail` Forces a failure. First, the machine pops any call entries from the top of the stack. Then, if the stack is empty, the machine stops and the whole pattern fails. Otherwise, the machine pops the top 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.

`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.)

`Call` ***label*** Calls instruction *label*; that is, saves in the call stack the address of the next instruction and jumps to instruction *label*.

`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.)

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

`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.)

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 theoretical 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.)

Now, let us see how patterns translate to programs for the parsing machine. In the following manipulations, we assume that all patterns always end with an `End` instruction, but this last instruction is always discarded when the pattern is used to compose other patterns.

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'
```

A number n translates to a sequence of n `Any` instructions. 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`. It receives `<p1>` and `<p2>` (the programs that represent patterns `p1` and `p2`) and returns the following program:

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

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, 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.

A more elaborated 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 more efficient than the first. For instance, when `<p1>` succeeds, the first code executes two `Choice` operations plus two commits, while the second code executes only one of each. The more alternatives we have, the worse left associativity becomes when compared to 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), so 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.

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 this sequence of two instructions:

```
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 same, so `PartialCommit` updates only the current subject position and the capture list length. With this instruction, the program looks like this:

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

LPEG provides a special 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, so this instruction never fails.)

An optional pattern, written as `p?`, is equivalent to `p + ''`:

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

Other optional patterns, such as `p{0,2}` (which matches `p` at most twice), could be translated to a sequence of unitary options:

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

A first optimization in this sequence comes from our knowledge that, once an item fails, all others will fail too (as they are all equal). There is no point in trying them. So, all alternative labels can point directly to the pattern's end:

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

Now, we have again the occurrence of the form `Commit L1; L1: Choice L2`. So, we can replace them by the `PartialCommit` instruction:

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

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 simply 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 charsets, 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:   ...
```

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 position and jumps to the pattern's end. If <p> fails, the control goes to L1, wherein the whole pattern fails.

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. So, 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.[7] 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 ')'
}
```

The resulting code is as follows:

```
      Call S
      Jump L2

S:    Choice L4
```

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

```
     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 easy 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`. So, 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
     Jump X
L3:  Return
L2:  ...
```

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

LPEG uses a conservative algorithm to avoid 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 [For04]. 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 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).

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[8]. When

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

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 to 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 worth 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 is 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 a little 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.

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
```

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 can 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 (e.g., the next character is not an `'n'`), 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 a 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, while the second option cannot start with an open parenthesis. Therefore, if the first `TestChar` succeeds, we know the second option cannot succeed. There is no need for a `Choice` instruction at all, because the given alternative will always fail.

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 a 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 PEG generates rules like this:

$$A \leftarrow a_1\ A_1 \ /\ \cdots\ /\ a_n\ A_n$$

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
```

But now the call became a tail call, so 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, so 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.

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. So, for instance, the code for `{'ana'}` 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's 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).

One way to avoid these pitfalls is to move the `Capture` instruction as far ahead 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.

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 it. 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, must satisfy 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 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.) So, let us have a look at more generic patterns.

The time to match a character (or any, or a charset) 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 (or both) components 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. So, 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. So, 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).

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 we have a recursive definition like `A <- ... A ...`. This pattern must consume at least one character before recursively calling `A` again—otherwise this infinite loop would be rejected by the compiler. So, if the input size for the first call is n, we can 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. So, 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 lower 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 one important exception, the ad hoc optimizations in the parsing machine do not change the complexity of matches; so, they do not affect the previous discussion. The important exception is tail call, that 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.

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 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 (Perl Compatible Regular Expressions). 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 0.7, Lua 5.1, PCRE version 6.7 04-Jul-2006, and the POSIX regex functions from the GNU C Library stable release version 2.5. The tests ran on a Linux machine (Ubuntu 7.04) with a Pentium 4 2.93 GHz and 1.5 GB of RAM.

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

Table 1: time (milliseconds) for searching a pattern in the Bible

For all tests, following Kernighan's lead [KW], we used as subject the full text of the King James Bible.[9] 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
```

(The pattern `(!p .)* p` gives similar results.) Moreover, we must quote all literals for LPEG, but not for the other tools. Table 1 shows the results. (For now, ignore the last column.) 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.[10] 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, while in LPEG we use an explicit loop (the tail-recursive grammar) for searching. With few (or none, as in the case with pattern `'@the'`) false starts, the search loop dominates the total time, and so 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-regex over LPEG decreases. (In the last entry, LPEG is faster than 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, so 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 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. With this search loop we get better results, presented in the last column ("LPEG(2)") of Table 1.

Table 2 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

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

[10] Unlike modern English texts, King James Bible has more 'h's than 'a's.

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 2: time (milliseconds) for searching 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

Table 3: time (milliseconds) for searching the last occurrence of a pattern in the Bible

13 letters. (The first such word in the text is "Jegarsahadutha", at line 2834.) The second pattern matches any word followed by the word "Abram". (It finds "begat Abram", at line 847.) The last pattern is similar, but changes "Abram" to "Joseph" (which appears later in the text, at line 2612). For all these patterns, LPEG has the fastest times.

Table 3 shows some results when searching for the last occurrence of a pattern in the subject. Now we are searching for the last occurrence of a pattern in the subject. As we explained in Section 3, 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)` to change the dot to match also newlines.) In LPEG we suffix the searching pattern with a `*`, so that the search repeats as many times as possible.

For some patterns (like a literal), PCRE optimizes in such a way that the search for the last occurrence of the pattern may be more than five times faster than the search for its 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 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 they 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 most 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 it like this:

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

With LPEG, every if statement with 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:

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

Table 4: time (milliseconds) for matching a pattern with the Bible

```
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 [Tis]. 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++). But even compiled C++ code does not compensate the overhead of memorization, as we can see in Table 4. 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, so 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 minutes.

6 Conclusion

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

LPEG presents some nice features and some novelties:

- It bases its patterns on a small set of primitives with formal semantics (Parsing Expression Grammars), 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 less than 2000 lines of C plus 120 lines of Lua. The main chunks in the C code are 220 lines of house-keeping code (type declarations plus some macros); 230 lines for the parsing machine per se; 130 lines for the infinite-loop verifier; 850 lines for the functions that create patterns (the "compiler"); and 260 lines for the processing of captures. As we already mentioned, the 120 lines of Lua implement the compiler for the regex syntax (using LPEG itself).

Although LPEG is written for Lua, it could be implemented easily for any scripting language. 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.

Acknowledgments

I would like to thank Mike Pall, for useful comments about initial versions of LPEG; Luiz Henrique de Figueiredo and Noemi Rodriguez, for useful comments about initial versions of this manuscript; and Francisco Sant'anna, for helping with some benchmarks. This work was partially supported by the Brazilian Research Council (CNPq, Grant #300993/2005-6).

References

[Abi] Abigail. Reduction of 3-CNF-SAT to Perl regular expression matching. `http://perl.plover.com/NPC/NPC-3SAT.html`.

[AU72] Alfred Aho and Jeffrey Ullman. Limited backtrack parsing algorithms. In *The Theory of Parsing, Translation and Compiling*, volume I: Parsing, chapter 6. Prentice Hall, 1972.

[CC97] Charles L. A. Clarke and Gordon V. Cormack. On the use of regular expressions for searching text. *ACM TOPLAS*, 19(3):413–426, 1997.

[For02] Bryan Ford. Packrat parsing: a practical linear-time algorithm with backtracking. Master's thesis, Department of Electrical Engineering and Computer Science, MIT, September 2002.

[For04] Bryan Ford. Parsing expression grammars: A recognition-based syntactic foundation. In *31st Symposium on Principles of Programming Languages (POPL'04)*. ACM, January 2004.

[Fow] Glenn Fowler. An interpretation of the POSIX regex standard. `http://www.research.att.com/~gsf/testregex/re-interpretation.html`.

[GPP71] R. E. Griswold, J. F. Poage, and I. P. Polonsky. *The SNOBOL4 Programming Language*. Prentice-Hall, Inc., second edition, 1971.

[IdFC96] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes. Lua—an extensible extension language. *Software: Practice and Experience*, 26(6):635–652, 1996.

[Ier] Roberto Ierusalimschy. LPeg — parsing expression grammars for Lua. `http://www.inf.puc-rio.br/~roberto/lpeg.html`.

[Ier06] Roberto Ierusalimschy. *Programming in Lua*. Lua.Org, Rio de Janeiro, Brazil, second edition, 2006. ISBN 85-903798-2-5.

[Knu71] Donald Knuth. Top-down syntax analysis. *Acta Informatica*, 1:79–110, 1971.

[KW] Brian W. Kernighan and Christopher J. Van Wyk. 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`.

[Lau] Ville Laurikari. TRE matching library. `http://laurikari.net/tre/`.

[Lau00] Ville Laurikari. NFAs with tagged transitions, their conversion to deterministic automata and application to regular expressions. In *Seventh International Symposium on String Processing and Information Retrieval (SPIRE'00)*, pages 181–187. IEEE, September 2000.

[Sha05] Y. Shafranovich. Common format and MIME type for Comma-Separated Values (CSV) files. RFC 4180 (Informational), October 2005.

[Tho68] Ken Thompson. Programming techniques: Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, 1968.

[Tis] Gordon Tisher. The Narwhal compiler suite. `http://sourceforge.net/projects/narwhal/`.

[WCO00] Larry Wall, Tom Christiansen, and Jon Orwant. *Programming Perl*. O'Reilly Media, third edition, 2000.