Mastering LPeg

by Roberto Ierusalimschy

(Version 1.1)

LPeg is a pattern-matching library for Lua, based on Parsing Expression Grammars. LPeg performs all tasks of a typical regex system, but it goes well beyond that. Among other tasks, we can write entire parsers with LPeg, with scanners included.

For us, *pattern matching* is a system for finding and extracting pieces of information from a text. For instance, we may want to find a line starting with "From:" in an email message and extract the rest of the line; we may have an XML document and want to extract all emphasized text, that is, text written between `<em>` and `</em>`; we may have a list of license plates and want to find all plates that are palindromes or that start with a given prefix. We may want to determine whether a sequence of characters is a valid identifier in C, that is, a letter or underscore followed by zero or more letters or underscores or digits; moreover, the sequence cannot be equal to a reserved word.

Most pattern-matching systems are based on *regexes*, also called *regular expressions*. (Most regex systems are extensions of the original regular-expression definition that break the nice properties of the original. For that reason, I prefer to save the name "regular expression" for the original definition and use the term "regex" for those extensions.) A regex is a string that specifies a pattern and occasionally what to extract from a *match*—an occurrence of that pattern in a text. As a simple example, consider the following Lua code:

```
subject = "birth date: 16/09/1998"
pattern = "(%d%d)/(%d%d)/(%d%d%d%d)"
d, m, y = string.match(subject, pattern)
print(d, m, y)  --> 16   09   1998
```

(Remember that, in Lua, a function can return multiple values.) In the pattern, `"%d"` represents any digit, `"/"` represents itself, and the parentheses delimit the *captures*, which is what to extract from the match. So, in this example, `pattern` means any two digits followed by a slash followed by two digits followed by another slash followed by four digits, capturing the three groups of digits. The function `string.match` searches for that pattern in the subject; if if finds a match, it returns the captured values, that is, the parts of the subject that matched the parenthesized parts of the pattern.

Unlike most other pattern-matching systems, LPeg is not based on regexes. Following the Snobol tradition, LPeg defines patterns as first-class objects in the language. This means that patterns are handled like regular Lua values. The LPeg library offers several functions to create and compose patterns; with the use of metamethods, several of these functions are provided as infix or prefix operators. On the one hand, the result is usually much more verbose than the typical encoding of patterns using regexes. On the other hand, first-class patterns allow us to create patterns piecemeal; it is easy to test each piece independently, to properly document them with good names and comments, to reuse those pieces, and to compose them to create more complex patterns. In other words, we can create patterns using the same conceptual tools we use for regular programming.

From a formal perspective, LPeg is not based on regular expressions plus a bunch of ad hoc extensions. Instead, LPeg is firmly grounded on *Parsing Expression Grammars* (PEGs), an established formalism with a well-defined semantics that are strictly more powerful than LL(k) parsers.

The Basics

Let us start with a very simple but complete Lua program that builds and uses a pattern:

```
local lpeg = require "lpeg"

-- creates a pattern
local p = lpeg.P("hello")

-- matches a subject against it
print(lpeg.match(p, "hello world"))        --> 6
print(lpeg.match(p, "hi world"))           --> nil
```

The function `lpeg.P` is the main function for creating patterns. When given a string, it returns a pattern that matches that string literally. After creating the pattern `p`, the code calls the function `lpeg.match` to match a given subject against the pattern. If the match succeeds, the call returns the position after the last character that matched. Otherwise, the call returns nil.

We can also call `match` as a method of the pattern:

```
print(p:match("hello world"))        --> 6
```

(Remember that Lua uses a colon to call methods.) The resulting position in a successful match is not particularly useful; usually what is relevant is whether the match succeeded or not. Later we will see several ways to extract more relevant information from a match.

Unlike what is common in traditional regex libraries, the method `match` does not search for a pattern. It tries to match only at the first position of the subject, doing the equivalent of what is called an *anchored* match in regex libraries. (That implies that, in case of a successful match, the returned value is always the number of matched characters plus one.) Later we will see how to search for a pattern in a subject.

The function `lpeg.P` does not handle any magic characters; all characters represent themselves in a string. For instance, the pattern created by`lpeg.P("%d")` matches a percent sign followed by the letter `d`. LPeg provides other functions to create character classes, repetitions, and the like.

From now on, our examples will assume that the library `lpeg` is already loaded in a variable `lpeg`. To test the examples, you can call the stand-alone Lua interpreter with the option `-llpeg` to pre-require the library into a global variable.

The function `lpeg.S` (`S` for *set*) receives a string and returns a pattern that matches a single occurrence of any character in that string. For instance, `lpeg.S"aeiou"` creates the set of vowels:

```
print(lpeg.S"aeiou":match("hello")) --> nil
print(lpeg.S"aeiou":match("all"))   --> 2
```

(Remember that, in Lua, we can omit the parentheses when calling a function with a single literal string as its argument.) The function `lpeg.R` (`R` for *range*) receives a two-character string and returns a pattern that matches a single occurrence of any character between those two characters:

```
print(lpeg.R"09":match("hello")) --> nil
print(lpeg.R"09":match("42"))    --> 2
```

More generally, `lpeg.R` can be called with several intervals, each represented by a two-character string. For instance, `lpeg.R("az","AZ")` matches any Latin letter, both lowercase and uppercase, and `lpeg.R("af","AF","09")` matches any hexadecimal digit.

In these last examples, the code creates the pattern and immediately applies `match` to it. This style is quite appropriate for presenting small examples, but you should seldom see it in real programs. Building a pattern in LPeg is much more expensive than using it, so usually a program first creates the patterns it will need and then uses them repeatedly in the rest of the code.

Similarly, in well-written programs those patterns probably should be kept in local variables, visible only in the file that defines and uses them. However, to help those that want to run these small examples directly in an interactive Lua session, I will use mostly global variables so that the code can be copied as is, directly to the interpreter prompt.

The function `lpeg.locale` creates a set of patterns according to the current locale, and returns them all in a table:

```
-- example using lpeg.locale
loc = lpeg.locale()
print(loc.space:match("  "))   --> 2
```

In the example, `loc` is the table containing those patterns, and `loc.space` is the pattern that matches white-space characters: regular space, newline, tabs, and form-feed. Besides `spaces`, the table includes also `alnum`, `alpha`, `cntrl`, `digit`, `graph`, `lower`, `print`, `punct`, `upper`, and `xdigit`.

From now on, several examples will assume the presence of a variable `loc` initialized as above.

When called with a positive integer, `lpeg.P` returns a pattern that matches that many characters, whatever they are. In the next example, the pattern matches the first three characters of the subject:

```
print(lpeg.P(3):match("hello world"))      --> 4
```

In this one, the match fails, because the subject does not have any characters:

```
print(lpeg.P(1):match(""))      --> nil
```

LPeg also allows us to combine patterns to create new patterns. It uses overloading to provide several of its constructors as operators. As a first example of this use, the star operator (`*`) represents concatenation. For instance, consider the following pattern:

```
p = lpeg.P(3) * lpeg.P("hi")
```

It matches strings that has the substring `"hi"` starting at the fourth character:

```
print(p:match("my hi"))       --> 6
print(p:match("his hi"))      --> nil
```

Whenever one of the operands for a binary operator is a pattern, Lua calls the respective metamethod. If the other operand is not a pattern, the metamethod tries to convert it by applying `lpeg.P`. So, we could write that previous pattern in one of these slightly simpler forms:

```
p = lpeg.P(3) * "hi"
p = 3 * lpeg.P"hi"
```

The basic constructor `lpeg.P` also accepts booleans. The call `lpeg.P(true)` produces a pattern that always succeeds without consuming any input, while `lpeg.P(false)` produces a pattern that always fails. In particular, the patterns `lpeg.P(true)`, `lpeg.P("")`, and `lpeg.P(0)` are all equivalent, as are equivalent the patterns `lpeg.P(false)`, `lpeg.S("")`, and `lpeg.R("za")`.

A note about Unicode

Throughout this text, when we talk about "character" we actually mean "byte". We need to keep that in mind when handling a text in Unicode.

Because Lua uses UTF-8 for Unicode encoding, most parts of LPeg work for free with Unicode text. In particular, literals, concatenations, repetitions, and predicates work correctly. However, sets and ranges are restricted to ASCII characters. Similarly, a construction like `lpeg.P(1)` matches one byte, not one Unicode character.

There are ways to work around this limitation, and LPeg version 1.1 introduced a pattern to match Unicode ranges, but we will not cover that matter in this text.

Repetitions and Choices

A pattern raised to a positive integer `N` results in a new pattern that behaves like the original pattern repeated `N` or more times. For instance, `loc.space^0` is a pattern that matches zero or more white-space characters, and `loc.alpha^1` matches one or more letters:

```
p = loc.space^0 * loc.alpha^1
print(p:match("  hello"))     --> 8
print(p:match("hello"))       --> 6
print(p:match("  "))          --> nil
```

In the first example, the repeated spaces matched two spaces and the repeated letters matched the word. In the second example, the repeated spaces matched zero spaces and the repeated letters matched the word. In the third example, the repeated spaces matched two spaces, but there were no letters to match the second half of the pattern, which required at least one letter; so the match failed.

Repetitions in LPeg are always *possessive*. That means that they match as many characters as possible, regardless of what comes next. A pattern like `lpeg.P(1)^0*lpeg.P("a")` always fails, because the initial repetition will match the whole string, leaving nothing to match the trailing `"a"`. Possessive repetition can be contrasted to *greedy* repetitions, the standard in conventional regex systems like those provided by Perl or Python. Greedy repetitions also match as many characters as possible—but provided that the whole match succeeds.

Often, greedy and possessive repetitions give the same results. Often, we must restrict what is being repeated so that greedy and possessive repetitions give the same results. As an example, consider the problem of matching a subject that has at least two semicolons. In conventional regex systems, with greedy repetition, a pattern like `".*;.*;"` would do the task. In words, that pattern means "match zero or more arbitrary characters, then match a semicolon, then match zero or more arbitrary characters, then match another semicolon". With possessive repetition, however, the initial repetition would consume the entire subject—semicolons and all—so the match would always fail. Nevertheless, it is easy to correct this problem by restricting the repetition patterns to mean "match zero or more characters *except semicolons*":

```
-- 'noSC' matches any character except a semicolon
noSC = lpeg.P(1) - ";"
p = noSC^0 * ";" * noSC^0 * ";"
print(p:match("one;two"))    --> nil
print(p:match("one;two;"))   --> 9
```

(In the definition of `noSC`, the subtraction removes the semicolon from the set of all characters created by `lpeg.P(1)`. We will discuss the subtraction operation in detail later, when we discuss predicates.)

Even in systems with greedy repetition, it is a good idea to use more restricted patterns when possible, as they are more robust and often more efficient.

A pattern raised to a negative integer results in a new pattern that matches the original one *at most* that many times. In particular, a pattern raised to -1 means an optional element:

```
-- 'num' is an optional minus sign followed by
-- one or more digits
num = lpeg.P"-"^-1 * lpeg.R"09"^1
print(num:match("-134"))    --> 5
print(num:match("351"))     --> 4
```

The plus operator creates choices. For instance, the pattern `lpeg.P("one")+lpeg.P("two")` (or simply `lpeg.P("one")+"two"`) matches either "one" or "two". Like repetitions, choices in LPeg are always possessive; if the first alternative matches, Lpeg will not consider the second one, independently of what comes next. (In the original PEG formalism, the construction is called an *ordered choice.*) As a concrete example, consider the following code:

```
p = (lpeg.P"a" + "ab") * "c"
print(p:match("abc"))       --> nil
```

In a system based on regular expressions, the similar pattern `"(a|ab)c"` would match `"abc"`. In LPeg, however, once the alternative `lpeg.P("a")` succeeds, the second one is not considered. In particular, choices like the previous one, where the first alternative is a prefix of the second, are pointless in LPeg: If the subject starts with an `a`, the first option matches and the second is not considered; otherwise, both options fail.

Note that the order of the alternatives in LPeg is relevant, often essential. As we just saw, the pattern `lpeg.P"a"+"ab"` is equivalent to the simpler `lpeg.P"a"`, but the pattern `lpeg.P"ab"+"a"` has a different meaning:

```
p = (lpeg.P"ab" + "a") * "c"
print(p:match("abc"))       --> 4
print(p:match("ac"))        --> 3
```

The first match uses the first alternative `"ab"`. In the second match, however, `"ab"` fails, so LPeg tries the second one (`"a"`), which succeeds.

A good mental model for understanding Parsing Expression Grammars in general, and LPeg in particular, is to think of a machine trying to match each pattern in turn. (That is how Lpeg actually works, bar optimizations.) For each sub-pattern, either there is match and the machine advances zero or more characters in the subject, or the match fails.

For simple characters or character classes, either the current character in the subject matches and the machine advances to the next character, or the current character is not in the class and the match fails.

In a sequence `p*q`, the machine first tries `p` and, if that succeeds, it proceeds to match `q`. If either of them fails, the sequence fails. In a choice `p+q`, the machine also first matches `p`; if that succeeds, the match of the choice is complete. Otherwise, the machine tries `q`. In a repetition `e^0`, the machine keeps trying to match `e` until that fails.

A drawback of this semantics happens when the expression inside a repetition can succeed without consuming any character. In that case, the machine could enter an infinite loop. To prevent that behavior, LPeg forbids the creation of loops where the body can match an empty string:

```
-- an optional dot
opt = lpeg.P"."^-1

-- zero or more optional dots (invalid!)
loop = opt^0
  --> stdin:1: loop body may accept empty string
```

Suppose we try to match the subject `"a"` against `loop`. It would try to match `"a"` against `opt`; because `opt` accepts an optional dot, it would match `"a"` without consuming any input character. In turn, since its body succeeded, `loop` would continue trying to match, and everything would repeat indefinitely.

Of course, when the loop accepts zero repetitions (a zero exponent), there is no need for its body to accept the empty string. For instance, the previous example is trivially fixed by making `loop` a repetition of regular (non-optional) dots. When the loop demands at least one occurrence, it is enough to change it to accept zero occurrences and again change the body so that it does not accept the empty string.

Simple Captures

Captures are patterns that produce values. The simplest capture in LPeg is the *simple capture*, created with the function `lpeg.C`. It receives a pattern and returns a capture that produces the string that matched the pattern:

```
-- capture the first word in a subject
p = loc.space^0 * lpeg.C(loc.alpha^1)
print(p:match(" hello world"))    --> hello
```

Whenever a pattern produces values, `match` returns those values instead of its default result. In the last example, the pattern captured the string that matched the repetition of alphabetic characters, and that is the value that `match` returned.

As is usual in LPeg, if its parameter is not a pattern, `lpeg.C` tries to convert it by applying `lpeg.P`:

```
-- matches three characters in the subject,
-- capturing the third one
p = lpeg.P(2) * lpeg.C(1)
print(p:match("hello"))    --> l
```

A pattern may have multiple captures, which can produce multiple values:

```
-- captures two characters and then one more
p = lpeg.C(2) * lpeg.C(1)
print(p:match("hello"))    --> he    l
```

As the next example shows, captures can be nested. In that case, the results from the outer captures come first:

```
p = lpeg.C(lpeg.C(2) * 1 * lpeg.C(2))
print(p:match("hello"))    --> hello   he    lo
```

Each time a capture matches, it produces its values. Therefore, the number of values produced by a pattern may depend on the subject. For instance, consider the two similar patterns `lpeg.C(lpeg.P("a")^0)` and `lpeg.C(lpeg.P("a"))^0`. In the first one, there is a single capture enclosing a repetition, so a match always produces one single result with a (possibly empty) sequence of `"a"`s:

```
print(lpeg.C(lpeg.P"a"^0):match("aaa"))    --> aaa
print(lpeg.C(lpeg.P"a"^0):match(""))       --> (empty string)
```

In the second pattern, the capture itself is being repeated, and each time it matches, it produces a new value:

```
print((lpeg.C(lpeg.P"a")^0):match("aaa"))  --> a   a   a
print((lpeg.C(lpeg.P"a")^0):match(""))     --> 1  (no captures)
```

Similarly, the pattern `lpeg.C(lpeg.P"a"^-1)` always produces one single value, that may be `"a"` or the empty string. The pattern `lpeg.C(lpeg.P"a"^)-1`, on the other hand, may produce one single `"a"` or no values at all, because the capture itself is optional.

The function `lpeg.Cp` creates a *position capture*, a pattern that captures the current position in the subject where the match occurred:

```
p = loc.space^0 * lpeg.Cp()
print(p:match("  hello"))    --> 3

p1 = p * loc.alpha^1 * lpeg.Cp()
print(p1:match("hello world"))    --> 1   6
```

The next example produces the positions of all words in the subject:

```
space = loc.space^0
word = loc.alpha^1
p = (space * lpeg.Cp() * word)^0
print(p:match("hello my world"))    --> 1   7   10
```

The pattern inside the repetition in `p` matches zero or more spaces followed by one or more letters, capturing the current position before the letters.

The default result from `lpeg.match`, when the pattern has no captures, is equivalent to what we would have if the pattern had a position capture at its end.

Predicates

A key property of PEGs is that its matching algorithm is deterministic. Following the mental model that we introduced, for a given pattern and a given subject the machine has always only one possibility of how to proceed. This determinism allows an easy concept of negation for matching: the *negation* of a pattern matches a subject if and only if the pattern itself fails for that subject.

In LPeg, we negate a pattern with the unary minus operator. That operation is called a *not predicate*. For instance, the pattern `-lpeg.P"a"` matches any string that does not match `lpeg.P"a"`, that is, any string that does not start with `"a"`:

```
p = -lpeg.P"a"
print(p:match("hello"))  --> 1
print(p:match(""))       --> 1
print(p:match("abc"))    --> nil
```

A not predicate never consumes any input, because either the predicate fails or its enclosed pattern fails. For the same reason, a not predicate never produces any capture.

An interesting use of a not predicate is in the pattern `-lpeg.P(1)`. The pattern `lpeg.P(1)` succeeds when there is any character in the subject, so `-lpeg.P(1)` will succeed only when there is no character, that is, at the end of the subject. This pattern is particularly useful to ensure that a successful match consumed the entire subject:

```
-- matches only strings of digits
p = loc.digit^0 * -lpeg.P(1)
print(p:match("123"))   --> 4
print(p:match("123 "))  --> nil
```

When we call the function `lpeg.P` with a negative integer `-n`, it generates the pattern `-lpeg.P(n)`. Remembering that LPeg operations automatically apply `lpeg.P` to non-pattern operands, we can write the previous pattern like this:

```
-- matches only strings of digits
p = loc.digit^0 * -1
```

PEGs offer another predicate called an *and predicate*, denoted by the length operator (`#`) in LPeg. For any pattern `p`, `#p` is equivalent to `-(-p)`: The pattern `#p` succeeds if and only if `p` succeeds, but `#p` never consumes any input.

Another syntax for the not predicate in LPeg is `p-q`, which is translated to `-q*p`. We can read it as "matches `p` provided that `q` doesn't match". For character sets, this operation corresponds to set difference. As an example, the pattern `lpeg.R("az")-lpeg.S("aeiou")` matches any Latin lowercase consonant, and the pattern `1-loc.space` matches any character that is not a space.

Searching

The not predicate is the basis of a very simple way to search for a pattern in a string. Assume a generic pattern `p`, and consider the following pattern:

```
searchP = (1 - p)^0 * lpeg.Cp() * p
```

The loop body `(1 - p)` checks that `p` does not match at the current position in the subject and then matches one character, advancing to the next position. The loop will repeat those steps as much as possible, that is, until either the end of the subject (where the `1` will fail) or a position where `p` matches. After the loop and the position capture, the final `p` succeeds only if the search has found its goal.

```
p = lpeg.P("needle")
searchP = (1 - p)^0 * lpeg.Cp() * p

s = "looking for a needle in a haystack"
print(searchP:match(s))   --> 15

s = "looking for a prickle in a haystack"
print(searchP:match(s))   --> nil
```

Example: Identifiers in the real world

In most programming languages, an identifier is described along the lines of "one alphabetic character followed by zero or more alphanumeric character". However, that is only half of the story. The other half is that a valid identifier cannot be a reserved word. For instance, `while` is not a valid identifier in Lua, even though it is a sequence of alphabetic characters.

It is easy to translate the first half of the story to LPeg:

```
id = loc.alpha * loc.alnum^0
print(id:match("count1"))  --> 7       -- Ok
print(id:match("while"))   --> 6       -- oops
```

The second half is more subtle. A first attempt is to use the not predicate to exclude the reserved words from the results:

```
reserved = lpeg.P("while") + "if" + "then"   -- ...
id = id - reserved
print(id:match("count1"))  --> 7        -- Ok
print(id:match("while"))  --> nil       -- Ok
print(id:match("if"))  --> nil          -- Ok
print(id:match("iffy"))  --> nil        -- oops
```

Once we see the problem, it is not difficult to understand its cause. Clearly, `"iffy"` matches the pattern `"if"`. The problem is that the pattern `reserved` is too lax, accepting anything that starts with those words. We need to make sure that it only accepts those words as entire words. The not predicate helps here again. With it, we can make sure that `reserved` matches only complete words:

```
reserved = (lpeg.P("while") + "if" + "then") * -loc.alnum
```

This new definition only matches a reserved word if it is not followed by an alphanumeric character:

```
print(reserved:match("if"))     --> 3
print(reserved:match("if()"))     --> 3
print(reserved:match("iffy"))   --> nil
```

Now we can complete our pattern for identifiers:

```
id = (loc.alpha * loc.alnum^0) - reserved
print(id:match("count1"))  --> 7        -- Ok
print(id:match("while"))  --> nil       -- Ok
print(id:match("if"))  --> nil          -- Ok
print(id:match("iffy"))  --> 5          -- Ok
```

As a final touch, we can construct the pattern `reserved` programmatically. Instead of writing all reserved words into the pattern, we can have them in a list:

```
rw = {"if", "then", "else", "while", "do", }
```

Using this list, the following loop builds `reserved`:

```
reserved = lpeg.P(false)
for _, w in ipairs(rw) do
  reserved = reserved + w
end
reserved = reserved * -loc.alnum
```

Note the initialization with `lpeg.P(false)`, which is the neutral element for choices in PEGs.

Example: List of assignments

Now let us consider how to parse a list of assignments of the form `name=name` separated by commas; there can be spaces anywhere around each element. We start defining our basic elements:

```
S = loc.space^0       -- spaces
name = lpeg.C(loc.alpha^1) * S
comma = "," * S
eq = "=" * S
```

Note that each basic element takes care of the spaces following it. The spaces preceding the element are parsed by the previous element. (Note too that `name` avoids adding the spaces to its capture.) This simple schema takes care of all spaces, except those in the very beginning of the subject. For those, we will add an extra `S` to the beginning of the final pattern.

The next step is to define the assignments:

```
assg = name * eq * name
```

The handle of the commas is a little tricky. If we add a comma to the end of `assg`, any list will need a comma after the last assignment. If we make that comma optional, we would allow two assignments without a comma between them. The following definition avoids these corners:

```
list = assg * (comma * assg)^0
```

If we want to allow empty lists, we make the whole list optional:

```
list = (assg * (comma * assg)^0)^-1
```

Finally, we should not forget to allow spaces at the beginning:

```
list = S * (assg * (comma * assg)^0)^-1

s = " a = b , aaa=count, bb = list  "
print(list:match(s))
  --> a    b    aaa    count    bb    list
```

Aggregating Captures

When a match produces multiple captures, more often than not we will want to aggregate those results. For instance, consider the following pattern, which captures all words in a sentence:

```
p = (loc.space^0 * lpeg.C(loc.alpha^1))^0
print(p:match("hello my world"))
  --> hello    my   world
```

We cannot further process those words unless we aggregate them, for example collecting them in a list.

For simple cases, we can build the aggregation directly in Lua, like here:

```
all = {p:match("hello my world")}
for _, w in ipairs(all) do print(w) end
```

```
  --> hello
  --> my
  --> world
```

However, when the pattern has more structure, returning a flat list of all its captures loses that structure. The example from the previous section, about lists of assignments, illustrates this point. In that case, it is still possible to reconstruct the structure, as each assignment generates exactly two values. However, consider the following small change to that example, where the value being assigned can be optional:

```
assg = name * (eq * name)^-1
list = assg * (comma * assg)^0
```

In this case, each item in the list can generate one or two captures. It is literally impossible to reconstruct the structure of the original list looking only to the list of captures:

```
print(list:match("a=x, b=c"))  --> a  x  b  c
print(list:match("a=x, b, c")) --> a  x  b  c
```

To preserve the original structure, we should aggregate the data as we parse the subject. LPeg offers several constructors to that end. The most general is the *function capture*, denoted by a division operator. When matching an expression `p/f`, LPeg first matches `p`; then it calls `f` passing as arguments all captures produced by `p`; finally, the values returned by `f` become the final captures of the whole expression. If `p` produces no captures, its whole match is passed to `f`.

The next code fragment illustrates a function capture:

```
function upperQuote (w)
  return '"' .. string.upper(w) .. '"'
end

p = loc.space^0 * (loc.alpha^1 / upperQuote)
print(p:match("  hi"))   --> "HI"
```

In this case, as `loc.alpha^1` generates no captures, its entire match goes to the function.

The next example uses two function captures:

```
num = loc.digit^1 / tonumber
p = (num * "+" * num) / function (a,b) return a + b end
print(p:match("32+64"))   --> 96
```

The pattern `num` uses a function capture with the predefined Lua function `tonumber` to convert its capture result from string to number; then the pattern `p` uses a function capture with an anonymous function to produce the sum of the two numbers captured by its subpatterns.

LPeg gives no guarantees about when, or even if, the function in a function capture will be called. Therefore, these functions should not produce side effects, such as assignments to global variables.

Example: Simple arithmetic expressions

Now let us see a somewhat more complex example, which will allow us to discuss more techniques and also to motivate and introduce other LPeg constructs. Our problem is to parse and evaluate simple arithmetic expressions. We will start with something simpler, supporting only additions and subtractions, so that we do not have to worry about precedence. Later we will introduce multiplicative operators.

As is usual in many larger uses of LPeg, first we define some lexical elements. Again we use the technique of handling spaces in these basic elements, so that we do not need to worry about spaces later.

```
S = loc.space^0       -- spaces
num = (loc.digit^1 / tonumber) * S
opA = lpeg.C("+") * S    -- add operator
opS = lpeg.C("-") * S    -- sub operator
```

As we do not have precedence yet, for now an expression can be seen as a list of numbers separated by operators. Ignoring for a moment how to evaluate the expression, the following pattern would match expressions:

```
exp = S * num * ((opA + opS) * num)^0
```

In words: An expression is a number followed by zero or more sequences of an operator, which can be a plus or a minus, followed by a number.

We still have to add semantics to the expression. One simple way to do that is to collect everything into a list. We use a function capture to build a list with all captures:

```
exp = exp / function (...) return {...} end
```

(Remember that the `...` is the syntax in Lua for a variadic function. The expression `{...}` builds a list with all parameters received by the function.)

With this definition, we have the following result:

```
t = exp:match("34 + 89 - 23")
for _, w in ipairs(t) do print(w) end
  --> 34
  --> +
  --> 89
  --> -
  --> 23
```

The need to collect all captures from a match into a list is so common that LPeg offers a special capture to that end: the *table capture*, denoted by the function `lpeg.Ct`. With that function, we can define `exp` like here:

```
exp = lpeg.Ct(S * num * ((opA + opS) * num)^0)
```

Now, let us see how to process that list, which is a list of numbers separated by strings denoting operators. A simple way to fold this list is like this:

```
function foldBin (lst)
  local acc = lst[1]
  for i = 2, #lst, 2 do
    if lst[i] == "+" then
          acc = acc + lst[i + 1]
    else   -- must be "-"
          acc = acc - lst[i + 1]
    end
  end
  return acc
end

print(foldBin(t))   --> 100
```

The pattern ensures that the list has at least one initial number. The accumulator `acc` starts with that number. Then the `for` loop traverses the rest of the list in steps of two, and for each pair operation–number, it updates the accumulator accordingly.

With this function defined, we can redefine `exp` to call it after creating the list:

```
exp = lpeg.Ct(S * num * ((opA + opS) * num)^0) / foldBin
print(exp:match("9 + 10 -12"))    --> 7
```

With a touch of functional programming, we can make things more reusable. Instead of representing each arithmetic operator in the list with a string, we can represent the operator with a function that performs the operation. So, instead of representing an addition with the string `"+"`, we can represent it with the following function:

```
function add (a, b) return a + b end
```

Similarly, subtraction can be represented by this function:

```
function sub (a, b) return a - b end
```

In our example, instead of `{34,"+",89,"-",23}`, the resulting list would be `{34,add,89,sub,23}`. Once we have such a list, the definition of `foldBin` can be both simpler and more generic:

```
function foldBin (lst)
  local acc = lst[1]
  for i = 2, #lst, 2 do
    local op = lst[i]          -- get operation
    acc = op(acc, lst[i + 1])  -- apply operation
  end
  return acc
end
```

The missing step is to change the pattern to create that kind of list as the result of a match. To this end, we will use a *constant capture*. A constant capture, built with the function `lpeg.Cc`, creates a capture that produces a constant value without consuming any input. With its help, we can redefine the patterns that match the operators:

```
opA = lpeg.P("+") * lpeg.Cc(add) * S
opS = lpeg.P("-") * lpeg.Cc(sub) * S
```

The pattern `opA`, upon matching a plus operator, produces the function `add`; similarly, `opS` produces `sub`.

Like table captures, constant captures are not inherently necessary. We could achieve the same effect with a function capture, using a function that returns a constant value:

```
opA = (lpeg.P("+") / function () return add end) * S
```

Again like table captures, constant captures offers an easier and more efficient way to code a somewhat common task.

Multiplicative operators

Our next step is to add multiplicative operators to our expressions. Again, we start with the basics:

```
local function mul (a, b) return a * b end
local function div (a, b) return a / b end

local opM = lpeg.P("*") * lpeg.Cc(mul) * S
local opD = lpeg.P("/") * lpeg.Cc(div) * S
```

The pattern `opM` matches a multiplicative operator and produces the corresponding function `mul`, and `opD` does the same for division.

For the syntax, we could simply join the new operators to the old ones, like here:

```
exp = S * num * ((opA + opS + opM + opD) * num)^0
```

Without further changes, that implementation would not give us the correct precedence, because all operators are handled together. For instance, `"3+5*7"` would evaluate to 56, instead of 38. We could correct that in the function that folds the list, but it is easier to handle that kind of syntactic problem as a syntactic problem, that is, in the pattern itself.

To reflect the operators hierarchy, we will define two kinds of expressions. The first kind, *term*, represents expressions without additive operators. So, inside a term, we can have only multiplicative operators. The second kind of expression, *exp*, represents sums of terms. With these definitions, the expression `3+5*7` cannot be parsed as `(3+5)*7`, because we can only multiply numbers. However, it can be parsed as `3+(5*7)`, because both `3` and `5*7` form terms and we can add terms.

We express this idea in LPeg with the following definitions:

```
local term = lpeg.Ct(num * ((opM + opD) * num)^0) / foldBin
local exp = lpeg.Ct(term * ((opA + opS) * term)^0) / foldBin
```

To better understand how they work, consider the subject `"23*3-14/2"`. We start matching `exp`, which invokes `term`, which invokes `num`, which matches `"23"`. Then the loop inside `term` matches `"*"` followed by `"3"`, but it cannot match the `"-"`, so it stops there. Then, `term` calls `foldBin` and produces the number 69 as its capture. Back to `exp`, its loop matches `"-"` and calls `term` again. This time, `term` matches `"14/2"` and produces 7. When the loop in `exp` ends, its table capture has produced the list `{69,sub,7}`. Finally, `exp` calls `foldBin`, which will produce 62 as the final result of the match.

The following listing puts together all that we have discussed through this example, in the proper order.

```
local lpeg = require "lpeg"
local loc = lpeg.locale()

local function add (a, b) return a + b end
local function sub (a, b) return a - b end
local function mul (a, b) return a * b end
local function div (a, b) return a / b end

local S = loc.space^0       -- spaces
local num = (loc.digit^1 / tonumber) * S
local opA = lpeg.P("+") * lpeg.Cc(add) * S
local opS = lpeg.P("-") * lpeg.Cc(sub) * S
local opM = lpeg.P("*") * lpeg.Cc(mul) * S
local opD = lpeg.P("/") * lpeg.Cc(div) * S

local function foldBin (lst)
  local acc = lst[1]
  for i = 2, #lst, 2 do
    local op = lst[i]          -- get operation
    acc = op(acc, lst[i + 1])  -- apply operation
  end
  return acc
end

local term = lpeg.Ct(num * ((opM + opD) * num)^0) / foldBin
local exp = lpeg.Ct(term * ((opA + opS) * term)^0) / foldBin

exp = S * exp     -- allow spaces at the beginning

print(exp:match("34 + 89 * 23"))   --> 2081
```

A glaring omission in our syntax are parenthesized expressions. That construction demands recursion, because once we open a parentheses inside a term we are back to `exp`, which in turn may open another parenthesis, and so on. Up to now, however, we do not know how to write recursive patterns in LPeg. To that end, we will have to learn about grammars.

Grammars

Until now, we have covered only the “parsing expression” part of our underlying formalism, *Parsing Expression Grammars*. In this section we will cover the “grammar” part.

Our working example will be *S-expressions*, or *sexp* for short. S-expressions are the underlying syntax for Lisp, and it is a useful language on its own for describing nested lists and trees.

An sexp is either a name or a list of sexps enclosed in parentheses, like the next examples:

```
add
(add one two)
((name) (add one (two)) ab ())
```

Note that the description of this language is recursive. We cannot write a pattern for that language with what we have seen so far. To write such a pattern, we need recursion.

PEGs support recursion through grammars. Informally, a grammar is a set of named patterns where each pattern can refer to other patterns (or to itself) through their names. Using the standard PEG notation, we could write the following grammar for sexps:

```
sexp <- name / '(' list ')'
list <- (sexp spaces)*
```

(The `/` operator denotes ordered choice in standard PEG, corresponding to the `+` operator in LPeg.) The grammar follows quite closely our original description, spelling out the meaning of "list". In words, an sexp is either a name or a list enclosed in parentheses. A list, in turn, is zero or more sexps separated by spaces. (To keep the example simple, it does not include all spaces that it should.)

LPeg uses Lua tables to support grammars. For each pair key–value in the table, the key is the name of a pattern and the value is the corresponding LPeg pattern. To refer to a pattern, we use a *nonterminal* (or *variable*) pattern. The function `lpeg.V` receives the name of a pattern and returns the corresponding nonterminal pattern. This resulting nonterminal pattern matches exactly like the pattern it refers to. Because entries in a table have no intrinsic order, we need also a way to signal which pattern is the initial one. (In our example, we need to say that we want to match `sexp`, not `list`.) In LPeg, we do this by assigning either the initial pattern itself or its name to the index 1 of the table.

Following these instructions, the resulting grammar would look like this in LPeg:

```
name = loc.alpha^1
spaces = loc.space^0
g = {
  [1] = "sexp",
  sexp = name + "(" * lpeg.V"list" * ")",
  list = (lpeg.V"sexp" * spaces)^0
}
```

Once we have everything defined in the table, a call to the function `lpeg.P` *closes* the table and returns the final pattern:

```
p = lpeg.P(g)
print(p:match("(() (a (b c)))"))  --> 15
```

To "close" the table means to internally connect all nonterminals to their respective patterns, raising an error if there is any undefined nonterminal.

Remember that, in Lua, a table constructor assigns expressions without explicit keys to consecutive integers starting with 1. Remember also that, when calling a function with a sole table constructor as its argument, we can omit the parentheses. We can use these features to make the code slightly simpler and more idiomatic:

```
p = lpeg.P{"sexp",
  sexp = name + "(" * lpeg.V"list" * ")",
  list = (lpeg.V"sexp" * spaces)^0
}
```

The initial `"sexp"`, as it has no key, goes to the index 1 of the table, as intended.

Let us try that grammar in some examples:

```
print(p:match("(a (b) (c d))"))   --> 14
print(p:match("(a (b) (c d)"))    --> nil
print(p:match("a (b) (c d))"))    --> 2
```

The last example requires an explanation. LPeg matches prefixes. The whole string is not properly parenthesized, but its initial `"a"` is a name, which is a valid sexp; that prefix is what LPeg is matching.

It is easy to correct that glitch. A nice (and unusual) property of grammars in LPeg is that, once we close the table, the result is a pattern just like any other. In particular, we can use it as a building block in other patterns. To correct the glitch, all we have to do is to add a check for end-of-subject at the end of the pattern:

```
p = p * -1
print(p:match("(a (b) (c d))"))   --> 14
print(p:match("(a (b) (c d)"))    --> nil
print(p:match("a (b) (c d))"))    --> nil
```

As a final touch, we can add captures to our pattern. We add a simple capture for names and a table capture for lists:

```
p = lpeg.P{"sexp",
  sexp = lpeg.C(name) + "(" * lpeg.V"list" * ")",
  list = lpeg.Ct((lpeg.V"sexp" * spaces)^0)
}
```

After this change, a successful match will return a Lua structure that corresponds exactly to the subject's structure. For instance, the match `p:match("(a (b) (c d))")` will return the table `{"a",{"b"},{"c","d"}}`.

Searching with grammars

Grammars allow another quite simple way to search for a pattern inside a subject. Suppose we want to search for any pattern `p`. Now consider the following grammar:

```
searchP = lpeg.P{ p + 1 * lpeg.V(1) }
```

When a grammar has a single rule, it is convenient to leave the rule at index 1: It automatically becomes the initial rule and we do not need to write its index. With that in mind, the rule reads like this: “Try to match `p`; if that fails, advance one character and repeat." The next code fragment shows a concrete example:

```
p = lpeg.Cp() * "000"
searchP = lpeg.P{ p + 1 * lpeg.V(1) }
print(searchP:match("1230300300034"))    --> 9
```

Due to internal details of LPeg, this kind of search usually is faster than the search with the not predicate that we learned earlier.

Left Recursion

Like repetitions, recursive grammars also risk infinite loops. As an example, consider the following trivial grammar:

```
# warning: invalid LPeg code
p = lpeg.P{ lpeg.V(1) * "a" }
```

The grammar has only one rule, that first matches itself and then matches `"a"`. Applying our mental model to this pattern, it is easy to understand its problem: To match the rule, the machine first has to match the rule!

We say that a rule is *left recursive* whenever we can go from its start back to itself without advancing in the subject. To avoid infinite loops, LPeg forbids the creation of left-recursive rules:

```
p = lpeg.P{ lpeg.V(1) * "a" }
  --> stdin:1: rule '1' may be left recursive
```

Some cases of left recursion are obvious, such as when a rule starts invoking itself, but some cases can be subtler:

```
p = lpeg.P{ lpeg.P("a")^0 * lpeg.V(1) }
  --> stdin:1: rule '1' may be left recursive
```

Example: Parenthesized expressions

With grammars in our tool belt, we are ready to return to parenthesized expressions. In a parenthesized expression, each basic block—which were simple numerals until now—can be replaced by a full expression inside parentheses.

More concretely, the core of our pattern for expressions were like this:

```
term = lpeg.Ct(num * ((opM + opD) * num)^0) / foldBin
exp = lpeg.Ct(term * ((opA + opS) * term)^0) / foldBin
```

Now, `term` will be built from a new kind of expression, which is either a numeral or an expression enclosed in parentheses. We call that kind of expression a *primary* expression.

```
OP = "(" * S
CP = ")" * S
primary = num + OP * lpeg.V("exp") * CP
term = lpeg.Ct(primary * ((opM + opD) * primary)^0) / foldBin
```

Moreover, `exp` must be defined inside a grammar, to allow it to use itself recursively:

```
E = lpeg.P{"exp",
  exp = lpeg.Ct(term * ((opA + opS) * term)^0) / foldBin
}
```

You may be wondering why we do not need to put `primary` and `term` inside the grammar, too. The answer is that, in LPeg, a use of one pattern inside another works like a macro. The uses of `term` inside `exp` have exactly the same meaning as if we replaced them by the definition of `term`. A similar thing happens with the uses of `primary` inside `term`. Once we expand all of them, we end up with a rather complex definition of `exp` with instances of `lpeg.V("exp")` inside it. Those are the ones that will be closed when we call `lpeg.P`.

Although we do not need to put `primary` and `term` inside the grammar, it is customary to do so. Among other benefits, we group together the definitions of related things (different kinds of expressions) and reduce the internal size of the pattern. Following this custom, our grammar could be written like this:

```
local primary = lpeg.V("primary")
local term = lpeg.V("term")
local exp = lpeg.V("exp")
local E = lpeg.P({"exp",
    exp = lpeg.Ct(term * ((opA + opS) * term)^0) / foldBin,
    primary = num + OP * exp * CP,
    term = lpeg.Ct(primary * ((opM + opD) * primary)^0) / foldBin
})
```

The local variables `primary`, `term`, and `exp` are defined only for convenience, to avoid repeating the calls to `lpeg.V`. Any use of the variable `primary` can be replaced by its definition (`lpeg.V("primary")`), without any change in the resulting pattern.

Debugging and Error Reporting

In most systems, debugging a pattern can be a frustrating experience. When there is a match, all is well. But when a match fails, all we know is that it failed. As a failed match creates no captures, we have no idea what happened during the match.

In LPeg, we can ameliorate this problem with a *match-time capture*. Like a function capture, a match-time capture calls a supplied function to produce its values. Unlike a function capture, however, a match-time capture executes during the match—as implied by its name. Every time the machine matches that pattern, its function is immediately called. This behavior gives us a powerful tool for inspecting the machine's behavior during a match.

The function `lpeg.Cmt` creates match-time captures. It receives a pattern `patt` and a function `func`. In a first approximation, the resulting pattern is somewhat similar to the result of the function capture `patt/func`: The resulting pattern tries to match `patt` and, if it succeeds, it calls `func`. However, there are several important differences between a match-time capture and a function capture:

- As we already discussed, the function in a match-time capture is called at match time, whenever its corresponding pattern matches, independently of what happens later with the whole match.

- Before the captures produced by `patt`, `func` receives two extra arguments: The whole subject of the match and the current position.
- The function in a match-time capture has a say about whether the match succeeds: If the function returns true as its first result, the match succeeds; if it returns false, the match fails.

Often we use match-time captures over an empty pattern, with the sole goal of calling the function whenever the match reaches that capture. For these cases, we can create the pattern by calling `lpeg.P` with a function as the sole argument.

Let us see an example, just to make things a little more concrete. Suppose we have the following failure, and we are trying to understand what is going wrong:

```
print(("ab" * lpeg.R("cC")):match("abc"))   --> nil
```

(The example is rather naive, but it illustrates the relevant points.) To follow what is happening during the match, we first create a match-time capture:

```
I = lpeg.P(function (s,i)
  print(i)
  return true
end)
```

As the function always returns true, this pattern always succeeds, while also printing the current position in the match.

Now, we intercalate it with our original pattern:

```
p = I * "a" * I * "b" * I * lpeg.R("cC") * I
```

Finally, we match our subject against this new pattern and check the output:

```
print(p:match("abc"))
  --> 1
  --> 2
  --> 3
  --> nil
```

The first `I` printed `1`, the current position before the match started; the second `I` printed `2`, the current position after matching `"a"`; the third `I` printed `3`, the current position after matching `"b"`. But there was no output from the fourth `I`, meaning that the match did not arrive there. (The `nil` was the final result of the match.) So, we can be confident that the match failed between the third and the fourth `I`, that is, in the pattern `lpeg.R("cC")`. Because `"c"` comes later than `"C"` in ASCII, that range is empty. Probably we meant `lpeg.S("cC")`, which matches `"C"` regardless of case.

When patterns become more complex, it may be difficult to sort out which output came from where. In those cases, it may be helpful to print some identifier together with the current position, or even only the identifier, when reaching a checkpoint. To this end, I often define `I` as a factory function that, when called with a tag, creates a match-time capture that prints that tag:

```
I = function (tag)
```

```
      return lpeg.P(function ()
                 print(tag)
                 return true
               end)
end
```

With this new definition, our previous example could be rewritten like this:

```
p = I'A' * "a" * I'B' * "b" * I'C' * lpeg.R("cC") * I'D'

print(p:match("abc"))
  --> A
  --> B
  --> C
  --> nil
```

Error Reporting

Match-time captures can also help in error reporting for the user of a pattern. Recall our grammar for arithmetic expressions. The way we left it, a match against `E` reads the input as far as it can, without any hints about errors. As an example,

```
print(E:match"34 + (25 * 3)) - 4")  --> 109
```

The erroneous suffix `") - 4"` is simply ignored, without any warning or error. The way to fix that problem is to force the match to go all the way to the end of the subject:

```
E = E * -1
```

Now, at least the match reports the error, but with zero extra information:

```
print(E:match"34 + (25 * 3)) - 4")  --> nil
```

In this small example, it is easy to spot the extra closing parenthesis, but imagine finding the error in a program with dozens of lines.

A simple and effective heuristic to locate errors in PEG matches is to report how far in the input the match went before failing. To track this information, we add to some rules an appropriate match-time capture that updates a variable with the position where it has been called. In our example of arithmetic expressions, it is enough to add this capture to the rule `S` for spaces, as that rule is ubiquitous in the grammar. We can define `S` as follows:

```
MaxOffset = 0     -- keep the maximum offset so far
local S = loc.space^0 * lpeg.P(
            function (_, p)
              MaxOffset = math.max(MaxOffset, p)
              return true
            end)
```

Using that definition through the entire grammar, any match will leave at the variable `MaxOffset` the last position where LPeg tried to match spaces, which is basically how far it went in the input:

```
subject = "34 + (25 * 3)) - 4"
print(E:match(subject))  --> nil
print(MaxOffset)        --> 14
print(string.sub(subject, 1, MaxOffset))
  --> 34 + (25 * 3))
```

The Accumulator Capture

In our examples with arithmetic expressions, we repeatedly used a technique of capturing a list of values and then *folding* the list. Fold, also called *reduce* or *accumulate*, is the process of traversing a list to produce some single final value. In imperative languages, we typically implement it as we did with our `foldBin` function: The function first initializes an accumulator with a neutral element or with the first element of the list; then, for each new element in the list, it combines the accumulator with the new element and updates the accumulator with the result. At the end of the list, the accumulator has the final result. This process is quite common, so LPeg provides a special capture for it, properly called *accumulator capture*.

Before we introduce this capture, let us analyse more carefully what happens in a simple example that needs folding: arithmetic expressions with addition and subtraction.

The pattern itself is easy to define:

```
space = lpeg.S" \t"^0
number = lpeg.R("09")^1 / tonumber * space
op = lpeg.C(lpeg.S"+-") * space
exp = lpeg.Ct(number * (op * number)^0)
```

However, we don't want to create a list. Instead, we want to directly combine the captured values to produce the final result. More specifically, for each new number, we want to operate it with an accumulated total. How can we do that?

A key property of all captures we have seen so far is that each capture is self-contained: It only operates on values from other captures that occur nested inside it. If we respect this restriction to compute the result of the expression, we must use a capture encompassing the whole expression, as the table capture that we have been using. However, if we could lift that restriction, we could instead use a capture that operates *out of its box*, a capture that could manipulate captures not nested inside it. That is precisely what the *accumulator capture* does.

An accumulator capture is somewhat similar to a function capture, with two key differences. First, besides the values produced by nested captures, the function receives as its first argument the last value produced by any capture before itself. Second, instead of producing a new capture, the value returned by the function replaces that last value.

As a very simple example, consider the next code fragment, focusing on the function capture inside the pattern:

```
function aux (...) print(...); return "X" end
```

```
p = lpeg.C(1) * (lpeg.C(1) / aux)
print(p:match("abc"))
  --> b
      a    X
```

As expected, the function `aux` received only the capture nested inside the function capture; moreover, the whole match produced an `"a"`, from the first capture— which is outside the function capture— and the return of the function, `"X"`.

Now, let us replace the function capture by an accumulator capture, which is denoted by a `%`:

```
p = lpeg.C(1) * (lpeg.C(1) % aux)
print(p:match("abc"))
  --> a    b
         X
```

We see that, now, the function received the first capture—which is not nested inside the function capture—as its first argument, and the only result from the match was the `"X"` returned by the function.

The accumulator capture is exactly what we need to process our simple rule for arithmetic expressions. Consider again that rule, but with an added accumulator capture instead of a table capture:

```
exp = number * ((op * number) % fadd)^0
```

With this pattern, the function `fadd` will be called for each new match of the pair `op * number`. Unlike a function capture, the function `fadd` also receives the previously created capture value, and its result replaces that value. The function itself, which we must define before the pattern, can be as simple as this:

```
function fadd (a, op, b)
  if op == "+" then
    return a + b
  else   -- must be "-"
    return a - b
  end
end
```

Now, let us follow what happens when we match `exp` against the subject `"13+2-12"`. First, the initial pattern `number` matches and captures the number `13`. Then `op` matches `+` and the second `number` matches `2`, and so `fadd` will be called with these three arguments: The `+` and the `2` from its inner captures, preceded by the `13` from the previous capture. Then, the returned value, `15`, becomes the only result of the process until this point. Next, in the repetition, `op` matches `-` and the second `number` matches `12`. Note that the capture that occurred immediately before this second occurrence of the accumulator capture was the first occurrence of the accumulator capture! So, this time `fadd` will receive 15—the value produced by the previous capture—plus `"-"` and `12`, and then it will return 3, which becomes the only result from the match.

Using the accumulator pattern, we can directly create the final result of the expression, instead of creating a list and then traversing the list. The final code is simpler and more efficient.

Substitution Captures

Besides searching, most pattern-matching systems offer some substitution operation, that replaces all occurrences of a pattern inside the subject by something else. LPeg does not offer a function for this functionality; instead, LPeg performs substitutions through a *substitution capture*.

The function `lpeg.Cs` receives a pattern and creates a substitution capture. When the inner pattern has no captures on its own, the substitution capture behaves like a simple capture:

```
p = lpeg.Cs(loc.alpha^1)
print(p:match("hello"))    --> hello
```

A substitution capture gets interesting when there are other captures in its inner pattern. In this case, whenever a capture matches inside the substitution, the string that matched the pattern is replaced by the value produced by the capture.

For an example, suppose we have to capitalize several words in a text. Instead of capitalizing them one by one, we can mark each word to be capitalized—e.g., by enclosing it in braces—and write a script to do the job for us. For that script, we start by writing a pattern that captures a word enclosed in braces and produces its capitalization:

```
word = ("{" * lpeg.C(loc.alpha^1) * "}") / string.upper
word:match("{hello}")  --> HELLO
```

(The function `string.upper` is predefined in Lua.) Then, we use it in the following pattern:

```
p = lpeg.Cs((word + 1)^0)
print(p:match("a {big}, {big} step"))
  --> a BIG, BIG step
```

The pattern inside the substitution capture repeatedly matches either a word enclosed in braces or any character. As each word enclosed in braces and the braces themselves match inside a capture, they get replaced by the result of the capture, which is the word capitalized. The single characters are not inside any capture, so they are left unchanged.

LPeg offers two other captures that are particularly useful inside a substitution capture: *string captures* and *query captures*. Like a function capture, both string captures and query captures are denoted by the division operator. LPeg distinguishes the three by the type of the denominator.

A division by a string creates a string capture. Usually, a string capture produces the string itself. However, if the string contains a percent sign followed by a digit, then that escape sequence is replaced by the corresponding capture in the numerator pattern. To illustrate, consider the problem of reversing a list of pairs `name:value`; for each pair, we want to rewrite it as `value:name`. First, we need to match the pairs, capturing each component:

```
id = lpeg.C(loc.alpha^1)
pair = id * ":" * id
```

Then, to switch the components, we can use a capture string:

```
pair = pair / "%2:%1"
print(pair:match("key:value"))    --> value:key
```

In words, the result of the capture is the string "%2:%1" with "%2" replaced by the second capture of `pair` and "%1" replaced by the first capture.

With that part ready, we now define a list of pairs and enclose it in a substitution capture:

```
list = lpeg.Cs((pair * loc.space^0)^0)
subject = "a:alpha b:bravo c:Charlie"
print(list:match(subject))
--> alpha:a bravo:b Charlie:c
```

As a special case, the escape "%0" gets replaced by the entire string that matched the pattern:

```
p = lpeg.Cs((lpeg.P(1) / "%0.")^0)
print(p:match("hello world"))  --> h.e.l.l.o. .w.o.r.l.d.
```

Now let us see the query capture. It is built with the syntax `patt/table`. It behaves similarly to a function capture but, instead of calling a function, it queries a table to compute the final value of the capture. See the next example:

```
t = {["+"] = "plus", ["-"] = "minus"}
p = lpeg.Cs((lpeg.P(1) / t)^0)
print(p:match("4 + 3 - 2"))
--> 4 plus 3 minus 2
```

Note that all characters except "+" and "-" are left unchanged in the subject: When a capture is not present in the table, the query capture produces no values, and therefore the substitution capture ignores that capture.

If the pattern inside a query capture produces multiple captures, the first one is the index. As in a function capture, if the pattern produces no captures, its whole match is the index.

Example: Literal strings

As an advanced example using the substitution capture, let us consider the problem of reading a literal string from an hypothetical programming language and translating its escape sequences. To avoid confusion with the escape from Lua itself, our example will use a percent sign for its escape character.

The literal string must be written enclosed in single quotes. Inside it, a single quote closes the literal, an escape forms an escape sequence with its following character, and any other character represents itself. The valid escape sequences are "%n" (newline), "%t" (tab), "%'" (single quote), and "%%" (escape character). The final pattern should behave like this:

```
-- not implemented yet!
print(str:match("'hi%'%%%thi'"))  --> hi'%  hi
```

Now let us see how to implement that pattern. First, we define the escape and the quote characters:

```
ESC = "%"
Quote = "'"
```

Next, we define a table `t` that translates the second character of an escape sequence to its translation. We also define a set, named `escapes`, with these characters.

```
-- escape translations
t = {n = "\n", t = "\t", [Quote] = Quote, [ESC] = ESC}
-- Set of keys in 't'
escapes = lpeg.S"nt" + ESC + Quote
```

We then define two more sets: One with “magic” characters comprising escapes and quotes, and one with common (non magic) characters:

```
-- "magic" characters (escapes and quotes)
magic = lpeg.P(ESC) + Quote
-- common characters (anything but escapes and quotes)
CC = 1 - magic
```

The most relevant part comes now, the content of a string:

```
content = (CC + ESC * lpeg.C(escapes) / t)^0
```

It is a repetition of zero or more common characters or escape sequences. An escape sequence, as previously described, is composed by an escape followed by any character from the set `escapes`; this character is captured and the whole escape sequence produces the translation of the sequence, by querying the table `t`.

Our final pattern is composed by `content` inside a substitution capture and surrounded by quotes:

```
str = Quote * lpeg.Cs(content) * Quote
-- an example
print(str:match("'hi%'%%%thi'"))   --> hi'%  hi
```

The substitution capture replaces the escape sequences by their translations, as produced by the query captures; the common characters are left unchanged.

A nice property of the substitution capture is that it can have multiple nested captures. As an example, we can easily expand our literal-string processor to handle hexadecimal escaces. Hexadecimal escaces allow us to specify any character inside a string literal by giving its numeric code in hexadecimal. For instance, `%x27` denotes a single quote and `%x22` denotes a double quote. The following pattern matches those escapes and produces the corresponding character:

```
function hex2chr (h)
  return string.char(tonumber(h, 16))
end
hexesc = (ESC * lpeg.P"x" *
               lpeg.C(loc.xdigit * loc.xdigit) / hex2chr)
```

```
-- examples:
print(hexesc:match("%x27"))    --> '
print(hexesc:match("%x61"))    --> a
print(hexesc:match("%x7D"))    --> }
```

Now, we only have to include this pattern as another option in the contents:

```
esc = ESC * lpeg.C(escapes) / t
content = (CC + hexesc + esc)^0
str = Quote * lpeg.Cs(content) * Quote
print(str:match("'a %x27quoted%' word'"))
  --> a 'quoted' word
print(str:match("'%x54%x68%x65%x20%x45%x6e%x64'"))
  --> The End
```