LPeg

Parsing Expression Grammars For Lua, version 1.1

Introduction

LPeg is a new pattern-matching library for Lua, based on **Parsing Expression Grammars** (PEGs). This text is a reference manual for the library. For a more formal treatment of LPeg, as well as some discussion about its implementation, see **A Text Pattern-Matching Tool based on Parsing Expression Grammars**. (You may also be interested in my **talk about LPeg** given at the III Lua Workshop.)

Following the Snobol tradition, LPeg defines patterns as first-class objects. That is, patterns are regular Lua values (represented by userdata). The 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 the so called *regular expressions* (which typically are not regular expressions in the formal sense). On the other hand, first-class patterns allow much better documentation (as it is easy to comment the code, to break complex definitions in smaller parts, etc.) and are extensible, as we can define new functions to create and compose patterns.

For a quick glance of the library, the following table summarizes its basic operations for creating patterns:

Operator	Description
`lpeg.P(string)`	Matches `string` literally
`lpeg.P(n)`	Matches exactly `n` characters
`lpeg.S(string)`	Matches any character in `string` (Set)
`lpeg.R("xy")`	Matches any character between *x* and *y* (Range)
`lpeg.utfR(cp1, cp2)`	Matches an UTF-8 code point between `cp1` and `cp2`
`patt^n`	Matches at least `n` repetitions of `patt`
`patt^-n`	Matches at most `n` repetitions of `patt`
`patt1 * patt2`	Matches `patt1` followed by `patt2`
`patt1 + patt2`	Matches `patt1` or `patt2` (ordered choice)
`patt1 - patt2`	Matches `patt1` if `patt2` does not match
`-patt`	Equivalent to `("" - patt)`
`#patt`	Matches `patt` but consumes no input
`lpeg.B(patt)`	Matches `patt` behind the current position, consuming no input

As a very simple example, `lpeg.R("09")^1` creates a pattern that matches a non-empty sequence of digits. As a not so simple example, `-lpeg.P(1)` (which can be written as `lpeg.P(-1)`, or simply `-1` for operations expecting a pattern) matches an empty string only if it cannot match a single character; so, it succeeds only at the end of the subject.

LPeg also offers the **`re` module**, which implements patterns following a regular-expression style (e.g., `[09]+`). (This module is 270 lines of Lua code, and of course it uses LPeg to parse regular expressions and translate them to regular LPeg patterns.)

Functions

`lpeg.match (pattern, subject [, init])`

The matching function. It attempts to match the given pattern against the subject string. If the match succeeds, returns the index in the subject of the first character after the match, or the **captured values** (if the pattern captured any value).

An optional numeric argument `init` makes the match start at that position in the subject string. As in the Lua standard libraries, a negative value counts from the end.

Unlike typical pattern-matching functions, `match` works only in *anchored* mode; that is, it tries to match the pattern with a prefix of the given subject string (at position `init`), not with an arbitrary substring of the subject. So, if we want to find a pattern anywhere in a string, we must either write a loop in Lua or write a pattern that matches anywhere. This second approach is easy and quite efficient; see **examples**.

`lpeg.type (value)`

If the given value is a pattern, returns the string `"pattern"`. Otherwise returns nil.

`lpeg.version`

A string (not a function) with the running version of LPeg.

`lpeg.setmaxstack (max)`

Sets a limit for the size of the backtrack stack used by LPeg to track calls and choices. (The default limit is 400.) Most well-written patterns need little backtrack levels and therefore you seldom need to change this limit; before changing it you should try to rewrite your pattern to avoid the need for extra space. Nevertheless, a few useful patterns may overflow. Also, with recursive grammars, subjects with deep recursion may also need larger limits.

Basic Constructions

The following operations build patterns. All operations that expect a pattern as an argument may receive also strings, tables, numbers, booleans, or functions, which are translated to patterns according to the rules of function **`lpeg.P`**.

`lpeg.P (value)`

Converts the given value into a proper pattern, according to the following rules:

- If the argument is a pattern, it is returned unmodified.
- If the argument is a string, it is translated to a pattern that matches the string literally.
- If the argument is a non-negative number *n*, the result is a pattern that matches exactly *n* characters.
- If the argument is a negative number -*n*, the result is a pattern that succeeds only if the input string has less than *n* characters left: `lpeg.P(-n)` is equivalent to `-lpeg.P(n)` (see the **unary minus operation**).
- If the argument is a boolean, the result is a pattern that always succeeds or always fails (according to the boolean value), without consuming any input.
- If the argument is a table, it is interpreted as a grammar (see **Grammars**).
- If the argument is a function, returns a pattern equivalent to a **match-time capture** over the empty string.

`lpeg.B(patt)`

Returns a pattern that matches only if the input string at the current position is preceded by `patt`. Pattern `patt` must match only strings with some fixed length, and it cannot contain captures.

Like the **and predicate**, this pattern never consumes any input, independently of success or failure.

`lpeg.R ({range})`

Returns a pattern that matches any single character belonging to one of the given *ranges*. Each `range` is a string *xy* of length 2, representing all characters with code between the codes of *x* and *y* (both inclusive).

As an example, the pattern `lpeg.R("09")` matches any digit, and `lpeg.R("az", "AZ")` matches any ASCII letter.

`lpeg.S (string)`

Returns a pattern that matches any single character that appears in the given string. (The `S` stands for *Set*.)

As an example, the pattern `lpeg.S("+-*/")` matches any arithmetic operator.

Note that, if `s` is a character (that is, a string of length 1), then `lpeg.P(s)` is equivalent to `lpeg.S(s)` which is equivalent to `lpeg.R(s..s)`. Note also that both `lpeg.S("")` and `lpeg.R()` are patterns that always fail.

`lpeg.utfR (cp1, cp2)`

Returns a pattern that matches a valid UTF-8 byte sequence representing a code point in the range `[cp1, cp2]`. The range is limited by the natural Unicode limit of 0x10FFFF, but may include surrogates.

`lpeg.V (v)`

This operation creates a non-terminal (a *variable*) for a grammar. The created non-terminal refers to the rule indexed by `v` in the enclosing grammar. (See **Grammars** for details.)

`lpeg.locale ([table])`

Returns a table with patterns for matching some character classes according to the current locale. The table has fields named `alnum`, `alpha`, `cntrl`, `digit`, `graph`, `lower`, `print`, `punct`, `space`, `upper`, and `xdigit`, each one containing a correspondent pattern. Each pattern matches any single character that belongs to its class.

If called with an argument `table`, then it creates those fields inside the given table and returns that table.

`#patt`

Returns a pattern that matches only if the input string matches `patt`, but without consuming any input, independently of success or failure. (This pattern is called an *and predicate* and it is equivalent to *&patt* in the original PEG notation.)

This pattern never produces any capture.

`-patt`

Returns a pattern that matches only if the input string does not match `patt`. It does not consume any input, independently of success or failure. (This pattern is equivalent to *!patt* in the original PEG notation.)

As an example, the pattern `-lpeg.P(1)` matches only the end of string.

This pattern never produces any captures, because either `patt` fails or `-patt` fails. (A failing pattern never produces captures.)

`patt1 + patt2`

Returns a pattern equivalent to an *ordered choice* of `patt1` and `patt2`. (This is denoted by *patt1 / patt2* in the original PEG notation, not to be confused with the `/` operation in LPeg.) It matches either `patt1` or `patt2`, with no backtracking once one of them succeeds. The identity element for this operation is the pattern `lpeg.P(false)`, which always fails.

If both `patt1` and `patt2` are character sets, this operation is equivalent to set union.

```
lower = lpeg.R("az")
upper = lpeg.R("AZ")
letter = lower + upper
```

`patt1 - patt2`

Returns a pattern equivalent to *!patt2 patt1* in the origial PEG notation. This pattern asserts that the input does not match `patt2` and then matches `patt1`.

When successful, this pattern produces all captures from `patt1`. It never produces any capture from `patt2` (as either `patt2` fails or `patt1 - patt2` fails).

If both `patt1` and `patt2` are character sets, this operation is equivalent to set difference. Note that `-patt` is equivalent to `"" - patt` (or `0 - patt`). If `patt` is a character set, `1 - patt` is its complement.

`patt1 * patt2`

Returns a pattern that matches `patt1` and then matches `patt2`, starting where `patt1` finished. The identity element for this operation is the pattern `lpeg.P(true)`, which always succeeds.

(LPeg uses the `*` operator [instead of the more obvious `..`] both because it has the right priority and because in formal languages it is common to use a dot for denoting concatenation.)

`patt^n`

If `n` is nonnegative, this pattern is equivalent to $patt^{n}$ *patt**: It matches `n` or more occurrences of `patt`.

Otherwise, when `n` is negative, this pattern is equivalent to $(patt?)^{-n}$: It matches at most `|n|` occurrences of `patt`.

In particular, `patt^0` is equivalent to *patt**, `patt^1` is equivalent to *patt+*, and `patt^-1` is equivalent to *patt?* in the original PEG notation.

In all cases, the resulting pattern is greedy with no backtracking (also called a *possessive* repetition). That is, it matches only the longest possible sequence of matches for `patt`.

Grammars

With the use of Lua variables, it is possible to define patterns incrementally, with each new pattern using previously defined ones. However, this technique does not allow the definition of recursive patterns. For recursive patterns, we need real grammars.

LPeg represents grammars with tables, where each entry is a rule.

The call `lpeg.V(v)` creates a pattern that represents the nonterminal (or *variable*) with index `v` in a grammar. Because the grammar still does not exist when this function is evaluated, the result is an *open reference* to the respective rule.

A table is *fixed* when it is converted to a pattern (either by calling `lpeg.P` or by using it wherein a pattern is expected). Then every open reference created by `lpeg.V(v)` is corrected to refer to the rule indexed by `v` in the table.

When a table is fixed, the result is a pattern that matches its *initial rule*. The entry with index 1 in the table defines its initial rule. If that entry is a string, it is assumed to be the name of the initial rule. Otherwise, LPeg assumes that the entry 1 itself is the initial rule.

As an example, the following grammar matches strings of a's and b's that have the same number of a's and b's:

```
equalcount = lpeg.P{
  "S";   -- initial rule name
  S = "a" * lpeg.V"B" + "b" * lpeg.V"A" + "",
  A = "a" * lpeg.V"S" + "b" * lpeg.V"A" * lpeg.V"A",
  B = "b" * lpeg.V"S" + "a" * lpeg.V"B" * lpeg.V"B",
} * -1
```

It is equivalent to the following grammar in standard PEG notation:

```
  S <- 'a' B / 'b' A / ''
  A <- 'a' S / 'b' A A
  B <- 'b' S / 'a' B B
```

Captures

A *capture* is a pattern that produces values (the so called *semantic information*) according to what it matches. LPeg offers several kinds of captures, which produces values based on matches and combine these values to produce new values. Each capture may produce zero or more values.

The following table summarizes the basic captures:

Operation	What it Produces
`lpeg.C(patt)`	the match for `patt` plus all captures made by `patt`
`lpeg.Carg(n)`	the value of the n^th^ extra argument to `lpeg.match` (matches the empty string)
`lpeg.Cb(key)`	the values produced by the previous group capture named `key` (matches the empty string)
`lpeg.Cc(values)`	the given values (matches the empty string)
`lpeg.Cf(patt, func)`	folding capture (*deprecated*)
`lpeg.Cg(patt [, key])`	the values produced by `patt`, optionally tagged with `key`
`lpeg.Cp()`	the current position (matches the empty string)
`lpeg.Cs(patt)`	the match for `patt` with the values from nested captures replacing their matches
`lpeg.Ct(patt)`	a table with all captures from `patt`
`patt / string`	`string`, with some marks replaced by captures of `patt`
`patt / number`	the n-th value captured by `patt`, or no value when `number` is zero.
`patt / table`	`table[c]`, where `c` is the (first) capture of `patt`
`patt / function`	the returns of `function` applied to the captures of `patt`
`patt % function`	produces no value; it *accummulates* the captures from `patt` into the previous capture through `function`
`lpeg.Cmt(patt, function)`	the returns of `function` applied to the captures of `patt`; the application is done at match time

A capture pattern produces its values only when it succeeds. For instance, the pattern `lpeg.C(lpeg.P"a"^-1)` produces the empty string when there is no `"a"` (because the pattern `"a"?` succeeds), while the pattern `lpeg.C("a")^-1` does not produce any value when there is no `"a"` (because the pattern `"a"` fails). A pattern inside a loop or inside a recursive structure produces values for each match.

Usually, LPeg does not specify when (and if) it evaluates its captures. (As an example, consider the pattern `lpeg.P"a" / func / 0`. Because the "division" by 0 instructs LPeg to throw away the results from the pattern, it is not specified whether LPeg will call `func`.) Therefore, captures should avoid side effects. Moreover, captures cannot affect the way a pattern matches a subject. The only exception to this rule is the so-called ***match-time capture***. When a match-time capture matches, it forces the immediate evaluation of all its nested captures and then calls its corresponding function, which defines whether the match succeeds and also what values are produced.

`lpeg.C (patt)`

Creates a *simple capture*, which captures the substring of the subject that matches `patt`. The captured value is a string. If `patt` has other captures, their values are returned after this one.

`lpeg.Carg (n)`

Creates an *argument capture*. This pattern matches the empty string and produces the value given as the n^{th} extra argument given in the call to `lpeg.match`.

`lpeg.Cb (key)`

Creates a *back capture*. This pattern matches the empty string and produces the values produced by the *most recent* **group capture** named `key` (where `key` can be any Lua value).

Most recent means the last *complete outermost* group capture with the given key. A *Complete* capture means that the entire pattern corresponding to the capture has matched; in other words, the back capture is not nested inside the group. An *Outermost* capture means that the capture is not inside another complete capture that does not contain the back capture itself.

In the same way that LPeg does not specify when it evaluates captures, it does not specify whether it reuses values previously produced by the group or re-evaluates them.

`lpeg.Cc ([value, ...])`

Creates a *constant capture*. This pattern matches the empty string and produces all given values as its captured values.

`lpeg.Cf (patt, func)`

Creates a *fold capture*. This construction is deprecated; use an **accumulator pattern** instead. In general, a fold like `lpeg.Cf(p1 * p2^0, func)` can be translated to `(p1 * (p2 % func)^0)`.

`lpeg.Cg (patt [, key])`

Creates a *group capture*. It groups all values returned by `patt` into a single capture. The group may be anonymous (if no key is given) or named with the given key (which can be any non-nil Lua value).

An anonymous group serves to join values from several captures into a single capture. A named group has a different behavior. In most situations, a named group returns no values at all. Its values are only relevant for a following **back capture** or when used inside a **table capture**.

`lpeg.Cp ()`

Creates a *position capture*. It matches the empty string and captures the position in the subject where the match occurs. The captured value is a number.

`lpeg.Cs (patt)`

Creates a *substitution capture*, which captures the substring of the subject that matches `patt`, with *substitutions*. For any capture inside `patt` with a value, the substring that matched the capture is replaced by the capture value (which should be a string). The final captured value is the string resulting from all replacements.

`lpeg.Ct (patt)`

Creates a *table capture*. This capture returns a table with all values from all anonymous captures made by `patt` inside this table in successive integer keys, starting at 1. Moreover, for each named capture group created by `patt`, the first value of the group is put into the table with the group key as its key. The captured value is only the table.

`patt / string`

Creates a *string capture*. It creates a capture string based on `string`. The captured value is a copy of `string`, except that the character `%` works as an escape character: any sequence in `string` of the form `%`*n*, with *n* between 1 and 9, stands for the match of the *n*-th capture in `patt`. The sequence `%0` stands for the whole match. The sequence `%%` stands for a single `%`.

`patt / number`

Creates a *numbered capture*. For a non-zero number, the captured value is the n-th value captured by `patt`. When `number` is zero, there are no captured values.

`patt / table`

Creates a *query capture*. It indexes the given table using as key the first value captured by `patt`, or the whole match if `patt` produced no value. The value at that index is the final value of the capture. If the table does not have that key, there is no captured value.

`patt / function`

Creates a *function capture*. It calls the given function passing all captures made by `patt` as arguments, or the whole match if `patt` made no capture. The values returned by the function are the final values of the capture. In particular, if `function` returns no value, there is no captured value.

`patt % function`

Creates an *accumulator capture*. This pattern behaves similarly to a **function capture**, with the following differences: The last captured value before `patt` is added as a first argument to the call; the return of the function is adjusted to one single value; that value replaces the last captured value. Note that the capture itself produces no values; it only changes the value of its previous capture.

As an example, let us consider the problem of adding a list of numbers.

```
-- matches a numeral and captures its numerical value
number = lpeg.R"09"^1 / tonumber

-- auxiliary function to add two numbers
function add (acc, newvalue) return acc + newvalue end

-- matches a list of numbers, adding their values
sum = number * ("," * number % add)^0

-- example of use
print(sum:match("10,30,43"))   --> 83
```

First, the initial `number` captures a number; that first capture will play the role of an accumulator. Then, each time the sequence `comma-number` matches inside the loop there is an accumulator capture: It calls `add` with the current value of the accumulator—which is the last captured value, created by the first `number`— and the value of the new number, and the result of the call (the sum of the two numbers) replaces the value of the accumulator. At the end of the match, the accumulator with all sums is the final value.

As another example, consider the following code fragment:

```
local name = lpeg.C(lpeg.R("az")^1)
local p = name * (lpeg.P("^") % string.upper)^-1
print(p:match("count"))    --> count
print(p:match("count^"))   --> COUNT
```

In the match against `"count"`, as there is no `"^"`, the optional accumulator capture does not match; so, the match results in its sole capture, a name. In the match against `"count^"`, the accumulator capture matches, so the function `string.upper` is called with the previous captured value (created by `name`) plus the string `"^"`; the function ignores its second argument and returns the first argument changed to upper case; that value then becomes the first and only capture value created by the match.

Due to the nature of this capture, you should avoid using it in places where it is not clear what is the "previous" capture, such as directly nested in a **string capture** or a **numbered capture**. (Note that these captures may not need to evaluate all their subcaptures to compute their results.) Moreover, due to implementation details, you should not use this capture directly nested in a **substitution capture**. You should also avoid a direct nesting of this capture inside a **folding capture** (deprecated), as the folding will try to fold each individual accumulator capture. A simple and effective way to avoid all these issues is to enclose the whole accumulation composition (including the capture that generates the initial value) into an anonymous **group capture**.

`lpeg.Cmt(patt, function)`

Creates a *match-time capture*. Unlike all other captures, this one is evaluated immediately when a match occurs (even if it is part of a larger pattern that fails later). It forces the immediate evaluation of all its nested captures and then calls `function`.

The given function gets as arguments the entire subject, the current position (after the match of `patt`), plus any capture values produced by `patt`.

The first value returned by `function` defines how the match happens. If the call returns a number, the match succeeds and the returned number becomes the new current position. (Assuming a subject *s* and current position *i*, the returned number must be in the range *[i, len(s) + 1]*.) If the call returns **true**, the match succeeds without consuming any input. (So, to return **true** is equivalent to return *i*.) If the call returns **false**, **nil**, or no value, the match fails.

Any extra values returned by the function become the values produced by the capture.

Some Examples

Using a Pattern

This example shows a very simple but complete program that builds and uses a pattern:

```
local lpeg = require "lpeg"

-- matches a word followed by end-of-string
p = lpeg.R"az"^1 * -1

print(p:match("hello"))        --> 6
print(lpeg.match(p, "hello"))  --> 6
print(p:match("1 hello"))      --> nil
```

The pattern is simply a sequence of one or more lower-case letters followed by the end of string (-1). The program calls `match` both as a method and as a function. In both sucessful cases, the match returns the index of the first character after the match, which is the string length plus one.

Name-value lists

This example parses a list of name-value pairs and returns a table with those pairs:

```
lpeg.locale(lpeg)   -- adds locale entries into 'lpeg' table

local space = lpeg.space^0
local name = lpeg.C(lpeg.alpha^1) * space
local sep = lpeg.S(",;") * space
local pair = name * "=" * space * name * sep^-1
local list = lpeg.Ct("") * (pair % rawset)^0
t = list:match("a=b, c = hi; next = pi")
        --> { a = "b", c = "hi", next = "pi" }
```

Each pair has the format `name = name` followed by an optional separator (a comma or a semicolon). The `list` pattern then *folds* these captures. It starts with an empty table, created by a table capture matching an empty string; then for each a pair of names it applies `rawset` over the accumulator (the table) and the capture values (the pair of names). `rawset` returns the table itself, so the accumulator is always the table.

Splitting a string

The following code builds a pattern that splits a string using a given pattern `sep` as a separator:

```
function split (s, sep)
  sep = lpeg.P(sep)
  local elem = lpeg.C((1 - sep)^0)
  local p = elem * (sep * elem)^0
  return lpeg.match(p, s)
end
```

First the function ensures that `sep` is a proper pattern. The pattern `elem` is a repetition of zero of more arbitrary characters as long as there is not a match against the separator. It also captures its match. The pattern `p` matches a list of elements separated by `sep`.

If the split results in too many values, it may overflow the maximum number of values that can be returned by a Lua function. To avoid this problem, we can collect these values in a table:

```
function split (s, sep)
  sep = lpeg.P(sep)
  local elem = lpeg.C((1 - sep)^0)
  local p = lpeg.Ct(elem * (sep * elem)^0)   -- make a table capture
  return lpeg.match(p, s)
end
```

Searching for a pattern

The primitive `match` works only in anchored mode. If we want to find a pattern anywhere in a string, we must write a pattern that matches anywhere.

Because patterns are composable, we can write a function that, given any arbitrary pattern `p`, returns a new pattern that searches for `p` anywhere in a string. There are several ways to do the search. One way is like this:

```
function anywhere (p)
  return lpeg.P{ p + 1 * lpeg.V(1) }
end
```

This grammar has a straight reading: its sole rule matches `p` or skips one character and tries again.

If we want to know where the pattern is in the string (instead of knowing only that it is there somewhere), we can add position captures to the pattern:

```
local Cp = lpeg.Cp()
function anywhere (p)
  return lpeg.P{ Cp * p * Cp + 1 * lpeg.V(1) }
end

print(anywhere("world"):match("hello world!"))   --> 7   12
```

Another option for the search is like this:

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

Again the pattern has a straight reading: it skips as many characters as possible while not matching `p`, and then matches `p` plus appropriate captures.

If we want to look for a pattern only at word boundaries, we can use the following transformer:

```
local t = lpeg.locale()

function atwordboundary (p)
  return lpeg.P{
    [1] = p + t.alpha^0 * (1 - t.alpha)^1 * lpeg.V(1)
  }
end
```

Balanced parentheses

The following pattern matches only strings with balanced parentheses:

```
b = lpeg.P{ "(" * ((1 - lpeg.S"()") + lpeg.V(1))^0 * ")" }
```

Reading the first (and only) rule of the given grammar, we have that a balanced string is an open parenthesis, followed by zero or more repetitions of either a non-parenthesis character or a balanced string (`lpeg.V(1)`), followed by a closing parenthesis.

Global substitution

The next example does a job somewhat similar to `string.gsub`. It receives a pattern and a replacement value, and substitutes the replacement value for all occurrences of the pattern in a given string:

```
function gsub (s, patt, repl)
  patt = lpeg.P(patt)
  patt = lpeg.Cs((patt / repl + 1)^0)
  return lpeg.match(patt, s)
end
```

As in `string.gsub`, the replacement value can be a string, a function, or a table.

Comma-Separated Values (CSV)

This example breaks a string into comma-separated values, returning all fields:

```
local field = '"' * lpeg.Cs(((lpeg.P(1) - '"') + lpeg.P'""' / '"')^0) * '"' +
                    lpeg.C((1 - lpeg.S',\n"')^0)

local record = field * (',' * field)^0 * (lpeg.P'\n' + -1)

function csv (s)
  return lpeg.match(record, s)
end
```

A field is either a quoted field (which may contain any character except an individual quote, which may be written as two quotes that are replaced by one) or an unquoted field (which cannot contain commas, newlines, or quotes). A record is a list of fields separated by commas, ending with a newline or the string end (-1).

As it is, the previous pattern returns each field as a separated result. If we add a table capture in the definition of record, the pattern will return instead a single table containing all fields:

```
local record = lpeg.Ct(field * (',' * field)^0) * (lpeg.P'\n' + -1)
```

Lua's long strings

A long string in Lua starts with the pattern [=*[and ends at the first occurrence of]=*] with exactly the same number of equal signs. If the opening brackets are followed by a newline, this newline is discarded (that is, it is not part of the string).

To match a long string in Lua, the pattern must capture the first repetition of equal signs and then, whenever it finds a candidate for closing the string, check whether it has the same number of equal signs.

```
equals = lpeg.P"="^0
open = "[" * lpeg.Cg(equals, "init") * "[" * lpeg.P"\n"^-1
close = "]" * lpeg.C(equals) * "]"
closeeq = lpeg.Cmt(close * lpeg.Cb("init"), function (s, i, a, b) return a == b end)
string = open * lpeg.C((lpeg.P(1) - closeeq)^0) * close / 1
```

The open pattern matches [=*[, capturing the repetitions of equal signs in a group named init; it also discharges an optional newline, if present. The close pattern matches]=*], also capturing the repetitions of equal signs. The closeeq pattern first matches close; then it uses a back capture to recover the capture made by the previous open, which is named init; finally it uses a match-time capture to check whether both captures are equal. The string pattern starts with an open, then it goes as far as possible until matching closeeq, and then matches the final close. The final numbered capture simply discards the capture made by close.

Arithmetic expressions

This example is a complete parser and evaluator for simple arithmetic expressions. We write it in two styles. The first approach first builds a syntax tree and then traverses this tree to compute the expression value:

```
-- Lexical Elements
local Space = lpeg.S(" \n\t")^0
local Number = lpeg.C(lpeg.P"-"^-1 * lpeg.R("09")^1) * Space
local TermOp = lpeg.C(lpeg.S("+-")) * Space
local FactorOp = lpeg.C(lpeg.S("*/")) * Space
local Open = "(" * Space
local Close = ")" * Space

-- Grammar
local Exp, Term, Factor = lpeg.V"Exp", lpeg.V"Term", lpeg.V"Factor"
G = lpeg.P{ Exp,
  Exp = lpeg.Ct(Term * (TermOp * Term)^0);
  Term = lpeg.Ct(Factor * (FactorOp * Factor)^0);
  Factor = Number + Open * Exp * Close;
}

G = Space * G * -1

-- Evaluator
function eval (x)
  if type(x) == "string" then
    return tonumber(x)
  else
    local op1 = eval(x[1])
    for i = 2, #x, 2 do
      local op = x[i]
      local op2 = eval(x[i + 1])
      if (op == "+") then op1 = op1 + op2
      elseif (op == "-") then op1 = op1 - op2
      elseif (op == "*") then op1 = op1 * op2
      elseif (op == "/") then op1 = op1 / op2
      end
    end
    return op1
  end
end

-- Parser/Evaluator
function evalExp (s)
  local t = lpeg.match(G, s)
  if not t then error("syntax error", 2) end
  return eval(t)
end

-- small example
print(evalExp"3 + 5*9 / (1+1) - 12")   --> 13.5
```

The second style computes the expression value on the fly, without building the syntax tree. The following grammar takes this approach. (It assumes the same lexical elements as before.)

```
-- Auxiliary function
function eval (v1, op, v2)
  if (op == "+") then return v1 + v2
  elseif (op == "-") then return v1 - v2
  elseif (op == "*") then return v1 * v2
  elseif (op == "/") then return v1 / v2
  end
end

-- Grammar
local V = lpeg.V
G = lpeg.P{ "Exp",
  Exp = V"Term" * (TermOp * V"Term" % eval)^0;
  Term = V"Factor" * (FactorOp * V"Factor" % eval)^0;
  Factor = Number / tonumber + Open * V"Exp" * Close;
}

-- small example
print(lpeg.match(G, "3 + 5*9 / (1+1) - 12"))   --> 13.5
```

Note the use of the accumulator capture. To compute the value of an expression, the accumulator starts with the value of the first term, and then applies `eval` over the accumulator, the operator, and the new term for each repetition.

Download

LPeg **source code**.

Probably, the easiest way to install LPeg is with **LuaRocks**. If you have LuaRocks installed, the following command is all you need to install LPeg:

```
$ luarocks install lpeg
```

License