LPEG for TEXies

It is not too hard!

Taco Hoekwater

One of the embedded Lua modules in LuaTEX is called LPEG. This article is an introduction to LPEG for new users. It assumes some passing knowledge of Lua as a programming language but treats LPEG as a completely new subject.

While explaining the basics of LPEG, we will create the beginnings of a streaming XML parser.

1. What is LPEG?

From the LPEG website[1] comes this quick definition:

> 'LPEG is a new pattern-matching library for Lua, based on Parsing Expression Grammars (PEGs).'

It is one of these sentences where you either have an 'oh, really?' moment or an 'ah, right!' moment depending on whether you already know what a PEG actually is.

The name LPEG is an acronym of 'Lua Parsing Expression Grammars'. The LPEG library does not use the exact syntax proposed in the original PEG article[2], but it is as close as it can get given the constraints of the Lua language.

The most important word in the expanded acronym is 'Grammar': LPEG works on the assumption that whatever you are trying to parse makes sense grammatically, for some to-be-defined grammar.

Creating a LPEG parser means building up a specialised grammar that can interpret your input. LPEG offers a mix of overloaded Lua operators and special functions that allow you to break up that input into grammatically sensible parts.

Because LPEG assumes a (strict) grammar governing the data it processes, it is ideally suited for parsing computer-readable and computer-generated text formats like XML documents, csv databases, (compiled) program source code, `ini` and other configuration files, as well as the save files of software programs that store their data as text files (like `fontforge`).

---

[1] http://www.inf.puc-rio.br/~roberto/lpeg/

[2] (if you are interested: https://bford.info/pub/lang/peg/)

On the flip side, LPEG is less suited for formats that have built-in ambiguities (like natural language texts) or that can alter their grammar on the fly (like TEX and some scripting languages).

2. Important LPEG functions and operators

There are quite a number of functions and operators defined by the LPEG module. Some are really basic, others are quite specialized. This article will only use the most basic functionality but as we will see, there is more than enough offered to already do very useful work.

All the functions in the LPEG library either operate on or create a Lua 'userdata' object.

A userdata object is an object that extends the Lua programming language by adding a new type of variable. Just like Lua variables can be of type `number`, `string`, or `table`, they can be of type `userdata`. Internally in Lua, there are many subtypes of userdata possible, each controlled by its own Lua library.

The LPEG library operates on the userdata objects that it creates itself. Each single one of those userdata objects deals with a specifically defined grammar. When someone (in this article or elsewhere) refers to 'an LPEG', what they mean is an instance of one those LPEG userdata objects that deals with a particular type of input.

We will see later how this all works in practice, but for now it is best to just show the list of functions, argument types, and operators that will be used in this article, as a reference for you to scroll back to as needed.

LPEG building block functions:

Function	Description
`lpeg.P(n)`	Matches exactly **n** characters
`lpeg.P(str)`	Matches `string` **str** literally
`lpeg.S(str)`	Matches any character in `string` **str** (Set)

LPEG object operators:

Operator	Description
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

LPEG matching operators and functions:

Function	Description
`lpeg.match(patt,input)`	Match **patt** against some **input**
`lpeg.C(patt)`	Match for **patt** plus all captures made by **patt**
`lpeg.Ct(patt)`	Table with all captures from **patt**
`patt / function`	Returns of function applied to the captures of **patt**
`patt / table`	`table[c]`, where **c** is the (first) capture of patt

3. First steps

As I wrote earlier, creating a LPEG means building up a specialized grammar that can interpret your input.

Let's create a first example of an LPEG, and even use it to match it against a string:

```
prefix = lpeg.P("hello world")

print('result:',lpeg.match(prefix,"hello world!"))
```

This prints the following to the terminal when run:

```
result:       12
```

There are two LPEG functions used in the example.

First `lpeg.P`: When this function is used with a string argument, it will create an LPEG object that matches exactly that string. By itself, that does not appear to be very useful. But we will see later that this is very handy while constructing more complicated LPEG objects.

Second `lpeg.match`: this is the function that actually tries to match your LPEG against an input string. It returns either the string index immediately following the successful match in the input string, or `nil` if there is no match. In this case, it returns 12 since string indices in Lua start at one and the LPEG created from the single `lpeg.P("hello world")` call produces a match that is 11 bytes long.

Now for a second experiment with the same LPEG:

```
print('result:',lpeg.match(prefix,"brave hello world"))
```

This prints the following to the terminal when run:

```
result:       nil
```

Here is something important to remember about LPEG: it always matches from the start of the input string, and it only succeeds if the whole LPEG can be matched. If you are used to traditional string searches or regular expressions, you will occasionally forget that and be completely surprised when it just 'doesn't work!'. That has happened to me numerous times when I started using LPEG. Do not give up, you will get used to this behaviour eventually.

It should be clear now why I gave my LPEG the variable name **prefix**: it is only good for finding exact prefixes in strings.

So, let's create a new LPEG that matches **hello world** anywhere in the input. For that, the * operator is needed. The terse description in the table above said this:

> '**patt1 * patt2**: matches **patt1** followed by **patt2**'

What does this mean exactly?

The overloaded operator * works on two LPEG objects, and combines them into a single new LPEG object that matches the concatenation of the two separate objects. A first example of how this works could look like this:

```
search = lpeg.P("hello world")
leading = lpeg.P("brave ")
infix = leading * search

print('result:',lpeg.match(infix, "brave hello world"))
```

This works, in that it correctly reports `result: 18` on the terminal. But the challenge is to find **hello world** anywhere, not just '7 characters into the string immediately following the word 'brave' and a space'.

An easy way to get out of the explicit '**brave**' is to use the alternative call version of **lpeg.P**, using a numeric argument such that it matches any character an exact number of times. Like this:

```
search = lpeg.P("hello world")
leading = lpeg.P(6)
infix = leading * search

print('result:',lpeg.match(infix, "xxxxx hello world"))
```

Somewhat better already. But we can improve the LPEG enormously by using an occurrence modifier, the operator ^. From the table above:

> '**patt^n**: Matches at least **n** repetitions of **patt**'

Now I should warn you that we cannot say **lpeg.P(0)**, as that produces an error. There are some internal rules that LPEGs must follow, and the most important one is that any LPEG has to have the possibility of a proper success/fail comparison. Matching nothing at all would always succeed (or always fail, depending on how you look at the world), which is why it is disallowed. For the same reason you cannot use the empty string **lpeg.P("")** either.

What we *can* do is match a single character zero or more times. However, here is another rule that makes LPEG very different from regular expressions: there is no backtracking over already matched input. So, the following attempt fails:

```
search = lpeg.P("hello world")
leading = lpeg.P(1)^0
infix = leading * search

print('result:',lpeg.match(infix, "xxxxx hello world"))
```

because the **leading** pattern will eat the entire input string, making the **search** fail. And because the **search** fails, the whole combined LPEG fails as well. We could fix this by matching a single character a limited number of times:

```
search = lpeg.P("hello world")
leading = lpeg.P(1)^-6
infix = leading * search

print('result:',lpeg.match(infix, "xxxxx hello world"))
```

But this is still not a proper solution. What we want to achieve is: optionally match anything that is not **hello world**, followed by **hello world**. Luckily we can do that, but at first glance the solution will look weird:

```
search = lpeg.P("hello world")
leading = lpeg.P(1)
infix = (leading - search)^0 * search

print('result:',lpeg.match(infix, "xxxxx hello world"))
```

To understand this, it is important to really look closely at the definition of the overloaded **-** operator:

> '**patt1 - patt2**: matches **patt1** if **patt2** does not match'

The function of the overloaded minus operator is *negation*, not *subtraction*. The parentheses group the **leading - search** together into a single unit, that is then matched zero or more times.

What happens is this: the match function looks at the next bit of input, and when it sees that the **search** LPEGfails, it next tries the **leading** LPEG. That one will match until the end of the string is reached so the combined group succeeds (and has now 'eaten' a single byte of the input). Because of the **^0**, this process is repeated until either **search** succeeds, or until the end of the input string is reached and **leading** also fails.

Grouping with parentheses like this happens a lot in LPEG definitions. If there is no occurrence modifier attached to it (like the **^0** in the example), a single match attempt is performed. A missing modifier is *not* equivalent to **^1**, as the latter allows for more than one match to happen!

So why is there a trailing *** search**? The given example works fine without it?

Consider what would happen if the string to be found did not appear in the input string at all. In this case, the matches would go on until both the **search** and the

leading inside the parentheses failed. But since we used a 'zero or more' modifier on the parenthesised part, this would actually count as a successful match, and **lpeg.match** would signal success by returning a string index one higher than the length of the input string.

There is a more subtle reason for *** search** as well: in a more complicated LPEG you will need to explicitly skip past succeeded matches, or the next match will fail. With the input string above, after the **(leading - search)^0** completes, the cursor is at string index 7. If more tests on the input were needed, then we need to move past the **hello world** in the input string.

Before we go on with slightly more complicated examples, it is worth mentioning that there is a lot of syntactic sugar allowed in LPEG definitions. In this article I will not use them often, but if you look at other people's LPEG code it is useful to know that bare numbers and strings can be used in a lot of places instead of explicit **lpeg.P** calls. This is because the overloaded operators will convert bare numbers or strings into the equivalent **lpeg.P** call if only one of the operands is already an LPEG object.

In particular, you will see the construct **(1-patt)** a lot, as this is an often needed building block in larger LPEG expressions.

The last example above could equivalently be written as:

```
search = lpeg.P("hello world")
infix = (1 - search)^0 * search
print('result:',infix:match("xxxxx hello world"))
```

In fact, this is also possible:

```
infix = (lpeg.P(1) - "hello world")^0 * "hello world"
```

but this is not as common.

This concludes all we need to know about the **lpeg.P** function for this article. It is possible to pass **lpeg.P** different types of arguments and with those it behaves somewhat differently. But that functionality is not needed for simple LPEG programs like those in this article, so I will go no deeper. Once you understand everything in this article, you can visit the LPEG website and read the official documentation for all the advanced possibilities.

For now, let us extend the example a little bit more. Suppose we also want to check for capitalised versions of *hello world*: *Hello world* and *Hello World*.

There are two different possible approaches. The first is to alter the definition of **search** by using an overloaded operator that we have not used yet: **+**. From the table:

> '**patt1 + patt2**: matches **patt1** or **patt2** (ordered choice)'

Our updated example using this approach would look like this:

```
search = (lpeg.P("hello world") +
          lpeg.P("Hello world") +
          lpeg.P("Hello World"))
infix = (1 - search)^0 * search
print('result:',infix:match("Hello world"))
```

The overloaded **+** operator signals alternatives. A warning is needed here: those alternatives are ordered and the further alternatives are abandoned after a successful match has been found.

What this means in practice is that the example below will *always* match **Green**, and *never* match **Green tea**.

```
search = (lpeg.P("Green") +
          lpeg.P("Green tea"))
infix = (1 - search)^0 * search
print('result:',infix:match("Green tea"))
```

When listing alternatives, always make sure that the matches are listed in declining scale of strictness. The following would work much better:

```
search = (lpeg.P("Green tea") +
          lpeg.P("Green"))
infix = (1 - search)^0 * search
print('result:',infix:match("Green tea"))
```

The second approach to the extended **hello world** example is by using the function **lpeg.S** and some extra concatenations.

> '**lpeg.S(str)**: matches any character in **string str**'

With that approach, the example looks like this:

```
search = (lpeg.S("Hh") * lpeg.P("ello ") *
          lpeg.S("Ww") * lpeg.P("orld"))
infix = (1 - search)^0 * search
print('result:',infix:match("Hello world"))
```

Both solutions are valid. Which one to use is a matter of taste. Depending on your input data one of two is probably more suitable than the other.

Internally, LPEG is based on comparing literal 8-bit bytes. Because of this, LPEG is more suited to dealing with artificial grammars than it is to dealing with natural languages, as capitalization and UTF-8 encoding is usually not an issue in computer-generated data structures. Matching words with random upper- and lowercase characters in them is quite possible with LPEG, but as you can see, it does not look natural.

If I were doing the current example exercise for real, I would solve it like this:

```
search = lpeg.P("hello world")
infix = (1 - search)^0 * search
print('result:',infix:match(string.lower("Hello world")))
```

This concludes the simple example. We will talk about the remaining functions that are needed when they appear in the actual XML parser example below.

4. A simple XML parser

4.1 Introduction

Let's assume the existence of a very trivial XML input file (**demo.xml**):

```
<?xml version="1.0"?>
<!-- This is just a short demo -->
<root>
    <title class="bold">Test</title>
    <p>Hello<br/>world</p>
    <p>Hans &amp;amp; me</p>
</root>
```

and also, that we want to parse this file somehow and to do some processing on it. For this, we need an XML parser.

There are many different XML parsers in the world, but they can be split into two main types: DOM (Document Object Model) and SAX (Simple API for XML). Client code that uses an XML parser interacts with those two types very differently.

On the one hand, there are so-called DOM parsers. DOM parsers read the whole of the XML input in one go, and create a tree-based data structure that represents the entire input. Only after the whole parsing process has been done can client code interact with the created data structure.

The created internal memory structure could look something like this (ignoring the whitespace in-between tags that is only there for readability):

```
{
  {type = 'pi',
   data = 'xml version="1.0"'},
  {type = 'comment',
   data = ' This is just a short demo '},
  {type = 'tag',
   name = 'root',
   data = {
     {type = 'tag',
      name = 'title',
      attr = {class = 'bold'},
```

```
      data = {
          {type = 'data', data = 'Test'}
        }
      },
      {type = 'tag',
       name = 'p',
       data = {
          {type = 'data', data = 'Hello'},
          {type = 'tag',  name = 'br'},
          {type = 'data', data = 'world'},
         },
      },
      {type = 'tag',
       name = 'p',
       data = {
          {type = 'data', data = 'Hans & me'},
        }
      }
    }
  }
}
```

This style of XML parser makes it possible to easily keep track of the 'location' of XML elements within the tree structure and makes it easy to extract various bits of data from the structure. It even makes it possible to mutate the XML structure in place (for instance, creating a Table of Contents) by looking at all the sectioning tags and then inserting those at the front of the document.

On the negative side DOM parsers use a lot of memory for large documents.

On the other hand, there are the SAX parsers. SAX parsers do not create any kind of data structure, but instead they are event-driven. While parsing is in progress, they send events to the client code to handle the found input.

When processing that same XML example, it would send a list of events to the client software that look like this (again ignoring the whitespace between elements):

```
PI      'xml version="1.0"'
COMMENT ' This is just a short demo '
START   'root'
START   'title', { class = 'bold' }
DATA    'Test'
END     'title'
START   'p'
DATA    'Hello'
START   'br'
END     'br'
```

```
DATA    'world'
END     'p'
START   'p'
DATA    'Hans & me'
END     'p'
END     'root'
```

Since events are processed in the exact order they appear in the XML input, this style of XML parser makes it much harder to keep track of the current equivalent tree location. Also processing data out-of-order is hard because the required bits and pieces have to be stored by the client software. On the positive side SAX parsers use almost no memory, even for enormous documents, and they are generally a bit faster than DOM parsers.

The XML parser in ConTEXt is of the DOM type. This makes sense for general use, because if you use XML documents with ConTEXt you often have use bits of the XML input out-of-order, and the DOM parser makes this an easy process.

However, if you need to process a big XML with lots of repetition of an identical substructure (like for instance, a database dump), a SAX parser would make a lot more sense.

Since parsing XML makes for an interesting but not-too-complicated LPEG use, developing a simple SAX parser would make a good example.

4.2 Initial analysis of needs

Before starting work with LPEG, it makes sense to analyse what is actually required.

We will build a very simple streaming XML parser using LPEG.

It will only handle:

- XML start tags
- XML end tags
- XML empty tags
- character data
- comments `<!-- ... -->`
- processing instructions `<? ... ?>`

For a production quality XML parser, much more would be needed. For example, the parser would need to accept various input encodings (we will assume UTF-8), accept XML namespaces, handle XML entities (both numerical and named), take care of error handling, et cetera.

The single biggest drawback of our parser is that it gobbles up the whole XML input into a single (potentially large) Lua string before processing starts. The ConTEXt DOM parser does the same thing but a SAX parser really should not do that. In a

real application, we would read from the input file only as much as is needed to produce the next client event but doing so would complicate our planned parser unnecessarily by adding unrelated non-LPEG code.

Anyway, our partial list is enough to process simple, syntactically correct XML. And adding all those extra features would make the LPEG longer and harder to explain.

So, what do we need to do?

- We need to create a separate Lua module for ConTEXt.
- We need to write a Lua function to parse an XML file and send events to the client application.
- We need to come up with a way for the client application to register code for the events it wants to handle.
- Most importantly, for use by the parsing function, we need to write an LPEG that can process the above XML features.

Let's get the show on the road.

We will create the file `xmlparser.lua`. Listed below is its planned contents without any of the code needed for the actual LPEG-based XML parser. The actual LPEG parser will be stored in the variable **`parser`** within **`xmlparser.parse`**. That part of the file will be completed in the next section of this article.

```
-- we will claim the 'xmlparser' table
xmlparser = {}

-- xml event function stubs
xmlparser.starttag = function (tag, attributes) end
xmlparser.endtag   = function (tag) end
xmlparser.data     = function (data) end
xmlparser.comment  = function (comment) end
xmlparser.pi       = function (pi) end

-- actual xml parsing function
xmlparser.parse = function (filename)
   local f = io.open(filename, "rb")
   local filedata = f:read("a*")
   f:close()
   if filedata then
      local parser
      -- rest of function body to be filled in
      parser:match(filedata)
   end
end
```

On the client side, this Lua module will be used like so:

```
dofile "xmlparser.lua"

xmlparser.starttag = function (tag, attributes)
  if #(table.keys(attributes))>0 then
      table.print(attributes, "START " .. tag)
  else
      print("START " .. tag)
  end
end

xmlparser.endtag = function (tag)
  print("END " .. tag)
end

xmlparser.data = function (text)
  print("DATA [[" .. text .. ']]')
end

xmlparser.parse("demo.xml")
```

As you can see, the events are handled by redefining the function stubs in `xmlparser.lua`. The events the client does not want to handle are simply left out.

4.3 File syntax handling

Those of you who are programmers are probably familiar with the notion of bottom-up versus top-down programming. In bottom-up programming, you start with the smallest building blocks and create the problem solving program by combining these small blocks into ever bigger parts of the solution. Top-down programming starts on the other side: the actual problem is broken down into smaller bits that are subdivided and solved until the final solution is reached.

Traditionally, parsers are an example of bottom-up programming. Bytes are interpreted from the input source one at a time, then combined into building blocks like numbers, variable names and keywords. Combining these building blocks produces statements, combining statements may produce functions, and eventually the top-level is reached and the problem solved (i.e. the file is parsed).

With LPEG, I find that it is often easier to do some work bottom-up, but most of the work top-down. That is not a set rule and may depend on your personal preferences but I, myself, find it easier this way.

To get started on processing XML, let's do the bottom-up bit first.

XML files consist of stuff within **<** and **>** and stuff outside of these. In order to separate those two groups, it makes sense to define two simple LPEGs that are each just a single character. These will be our building blocks for the more complex LPEG we will create later.

```
local lt        = lpeg.P("<")
local gt        = lpeg.P(">")
```

The stuff outside of **<** and **>** can also contain entities like the `&` from the example. Those always start with an ampersand and end with a semicolon. So, let's define those two single-character LPEGs as well:

```
local amp       = lpeg.P("&")
local semi      = lpeg.P(";")
```

The stuff within **<** and **>** has its own rules. We can deal with processing instructions and comments later on because the syntax for those is very simple. There is no internal structure, just a special type of start and end.

But actual XML tags have internal rules. There are start tags, end tags, and empty tags. To differentiate these, the XML specification uses the forward slash character. So:

```
local slash     = lpeg.P("/")
```

Start tags can have optional attributes. Attributes (like **class="**`bold`**"**) are specified as a space-separated list of key-value assignments. The values can be surrounded by either single or double quote ASCII characters and the key is separated from the value by an equals sign:

```
local eq        = lpeg.P("=")
local qt        = lpeg.P('"')
local sq        = lpeg.P("'")
```

The final two bottom-up LPEGs are for the tag names themselves, and the spaces between tag names and attribute definitions. Both of those are a little bit more complex. For 'space separation', the XML specification actually defines a small set of acceptable characters:

```
local space     = lpeg.S(" \t\n")
```

The official XML rule for a tag name is complicated but by assuming that our input file is not using complicated tricks we can use this simple definition: any sequence of characters that does not include spaces, the greater-than, or a forward slash character.

For readability of my LPEG sources, I usually define a short function called **without**, so let's introduce that as well:

```
local function without(p)
  return lpeg.P(1)-p
end

local tagname   = without(space + gt + slash)^1
```

Note that the function **without** does not add any parentheses around the return value. This is not necessary because the expression **lpeg.P(1)-p** creates a single LPEG object that is then returned. Likewise, it does not need to add parentheses around the argument **p** because it receives a single variable, not the whole expression.
This does not mean that parentheses are not important at all. The normal priorities for the overloaded operators are still in effect so these two LPEG statements are *not* equivalent:

```
a = (lpeg.P(1) - lpeg("/")) * lpeg(">")
b = lpeg.P(1) - lpeg("/") * lpeg(">")
```

The LPEG **a** matches input that consists of: 1) a character that is not a slash; and 2) is followed by a greater-than sign so that it matches the pattern **"**x>**"**.
The LPEG **b** matches input that consists of a character that is not followed by a slash and subsequent greater-than sign thereby failing on **"**x>**"**.

Having defined all the low-level stuff that is immediately interesting, I find it easier to go back to the top at this point. We can define the **parser** LPEG provisionally:

```
local parser
-- rest of  parser body to be filled in
parser = (pi + comment + endtag + starttag + data )^1
```

Of course, since none of those five variables have been defined yet, the example will not compile. Lua will complain about

```
attempt to perform arithmetic on a nil value
```

when you have undefined LPEG variables (or variables that are accidentally not LPEG objects), you will typically see the message above, or a variant of:

```
lpeg-pattern expected, got xxxx.
```

If you need a quick placeholder for an undefined LPEG variable, you can usually use this syntax:

```
local pi = lpeg.P(false)
```

Using a boolean as argument to **lpeg.P** creates an object that always matches false (or true), without looking at the input. Using **true** as argument is not always possible because complete matches that return **true** on empty input are forbidden, which is why **false** is better.

Or, you could do some more programming first before trying to run the code, of course ...

4.4 Filling in the parser function body

We left the previous section with this incomplete definition:

```
parser = (pi + comment + endtag + starttag + data )^1
```

Let's work on the LPEG variables **pi** and **comment** first. We can do them together, because they are very similar:

```
local pi = lpeg.P("<?") * without(lpeg.P("?>"))^1 * lpeg.P("?>")
local comment = lpeg.P("<!--") * without(lpeg.P("-->"))^1
              * lpeg.P("-->")
```

I wrote those assignments on single lines because they are so simple, but of course it is also possible (and perhaps more readable) to do it like this:

```
local pistart      = lpeg.P("<?")
local pistop       = lpeg.P("?>")
local pibody       = without(pistop)^1
local commentstart = lpeg.P("<!--")
local commentstop  = lpeg.P("-->")
local commentbody  = without(commentstop)^1
local pi           = pistart * pibody * pistop
local comment      = commentstart * commentbody * commentstop
```

If we were just parsing the XML and throwing it away immediately, the above definitions would be good enough. But we want to pass the bit between the delimiters to the client application, and for that a little bit of extra work is needed. First, we need to 'capture' that middle bit and then, we need to somehow pass it on to the client side.

For capturing, we need the function **lpeg.C**. From the table again:

> '**lpeg.C(patt)**: the match for **patt** plus all captures made by **patt**'

That sounds cryptic but it just means that we can feed a pattern into **lpeg.C** as its argument, like so:

```
local pi       = pistart * lpeg.C(pibody) * pistop
```

This captures the input of the match **pibody** and stores it. If there were any embedded captures (like if there were any embedded **lpeg.C** calls), these will be kept as well and returned following the 'current' capture.

These captures are actual return values. If you had used **lpeg.C** in the earliest example of this article:

```
prefix = lpeg.C(lpeg.P("hello world"))
print('result:',lpeg.match(prefix,"hello world!"))
```

The terminal would report:

```
result:       hello world
```

Note that if there are actual return values, the normal string index return value from **lpeg.match** is suppressed.

And if you 'overdo' it, like this:

```
prefix = lpeg.C(lpeg.C(lpeg.P("hello world")))
print('result:',lpeg.match(prefix,"hello world!"))
```

You will get

```
result:       hello world   hello world
```

You might think I am being silly, but when your LPEGs get more complicated, nested captures become prevalent rather quickly. Sometimes that is helpful (if you want to capture a larger object as well as sub-objects therein) but often you will find that it accidentally happens because you are re-using a single LPEG variable in different situations. Just be aware that this can happen.

So, the captures are return values for **lpeg.match**. Fine, but that is not what we want in this case. Processing the whole XML and then feeding all the separate captures back to the client is not helpful. We want to process the XML one capture at a time. For that, there is the overloaded operator **/**:

> '**patt / function**: the returns of function applied to the captures of **patt**'

The function you specify at the right-hand side gets as its arguments all the captures of the LPEG pattern on the left-hand side. The return value(s) from that function then replaces the captures as the return values of **lpeg.match**.

In typical use, you specify a function that returns nothing, and you depend on its side-effects to reach your goal. Still using the **hello world** example, take a look at this:

```
prefix = lpeg.C(lpeg.P("hello world")) / print
print('result:',lpeg.match(prefix,"hello world!"))
```

on the terminal, this outputs two lines:

```
hello world
result:       12
```

The call to **print** on the first line has intercepted the capture and printed it to the terminal immediately. Since **print** does not have any return values, the print statement on the second line just returns the string index, as in the earlier examples.

Armed with this new knowledge, defining **pi** and **comment** in the XML parser we are building becomes really easy:

```
local pi      = pistart * lpeg.C(pibody) / xmlparser.pi * pistop
local comment = commentstart
              * lpeg.C(commentbody) / xmlparser.comment
              * commentstop
```

Remember that we had already defined **xmlparser.pi** and **xmlparser.comment** earlier. So the task of parsing processing instructions and comments is now officially finished.

The next simplest construct to handle are XML end tags.

```
local endtag = lt * slash * lpeg.C(tagname) * gt
               / xmlparser.endtag
```

Every variable in that line is already defined above. Do you see that the **/** is now on a different location in the line? I did that on purpose, to remind you that the overloaded operators behave exactly like they normally would. The left-hand side of the **/** in this line is *not* four LPEG objects. The ***** operators on the left have all been processed already, so what is left is a single nameless LPEG object with one capture. And that combined object is the left-hand side of the **/** operator.

If there were two separate **lpeg.C** calls on the left-hand side of the **/**, the function would receive both of the captures from the now combined LPEG object (assuming the matches succeed, of course). If you do not want that, you can use parentheses to change the relative priorities of the overloaded operators.

The remaining two bits of the parser are both a bit more complicated. Let's deal with actual textual data first because it needs some helper code that will be useful later.

A first attempt would be to define something similar to what we have already done:

```
local data     = lpeg.C(without(lt)^1) / xmlparser.data
```

For very, very trivial cases, that would work. But if the data contains XML entities (like the &**amp**; in our **demo.xml** example), the parser is supposed to replace them with their value.

That is why we need to add a bit of indirection:

```
local function dodata (data)
    xmlparser.data(fixentities(data))
end
local data     = lpeg.C(without(lt)^1) / dodata
```

Actually, the parser is supposed to replace character references as well (these look like entities except that they are numerical or hexadecimal, e.g.   or &#**A0**;). But we will only look at the the five predefined named entities for this article.

Of course we have to define **fixentities** and its helper code:

```
local entities = { gt = ">", lt = "<", amp = "&",
                   quot = '"', apos = "'" }
local function fixentities (data)
  local entity = amp *  lpeg.C(without(semi)^1) / entities * semi
  local plain  = lpeg.C(lpeg.P(1))
  return table.concat
           (lpeg.match(lpeg.Ct((entity + plain)^1),data))
end
```

Whoa! That's lots of new stuff here.

First off, the **entities** table is just a bit of data structure. The keys are the entity names, and the values are their replacement texts.

The **fixentities** function actually builds its own private LPEG and matches that against the **data** argument it receives.

The initial definition of the local **entity** and **plain** variables looked like this:

```
local entity = amp * without(semi)^1 * semi
local plain  = lpeg.P(1)
```

and with that, the match itself would be:

```
lpeg.match((entity + plain)^1,data)
```

but (of course) that did not return anything except a byte count.

The LPEG pattern we use is splitting the input into the names of entities (via **entity**) and other bytes (via **plain**). Somehow, we need to replace the entity names by their values, create a combined table, and concatenate the whole thing together.

Once we have a Lua table with the content we want, it is a simple matter of using **table.concat** without a second argument to create a single string again.

To create table from the list of captures (that we have yet to create), we can use **lpeg.Ct**. That is literally all it does: it takes all of the captures, and creates a single table from them. That is why the last line becomes this:

```
  return table.concat
          (lpeg.match(lpeg.Ct((entity + plain)^1),data))
```

Since we will be using a capture to replace the entities and we are planning to combine all captures into a table, we need to also capture the bytes that are *not* entity names. That is what the **lpeg.C** in this line is for:

```
local plain  = lpeg.C(lpeg.P(1))
```

The last line that we need to cover is the one that actually does the entity replacements:

```
local entity = amp *  lpeg.C(without(semi)^1) / entities * semi
```

What is new here is that we use the table in **entities** on the right side of the **/** instead of a function. That was not strictly required but it is a bit of a shortcut that is explicitly allowed by LPEG. The alternative code below would work just as well (although it would run a bit slower):

```
local entityfunc = function (a) return entities[a] or '' end
local entity = amp *  lpeg.C(without(semi)^1) / entityfunc * semi
```

This is not the most efficient way to process XML entities in data. In fact, in this case, a set of **string.gsub()**s would likely be faster. But had I done that, then I would not have had a good reason to explain the **/** followed by a table and **lpeg.Ct**. Both are important building blocks, even in fairly simple LPEGs.

The last LPEG needed is the one for XML start tags. Unsurprisingly, that is the most complicated object to process but we can make our lives a bit simpler by postponing the detailed processing. The match needed to find a start tag is quite simple: a start tag everything from a less-than character that is not followed by a forward slash (as that would be an end tag) up to the closing greater-than character.

So the part that goes into **parse** is quite simple:

```
local starttag = lt * lpeg.C(without(gt)^1) * gt/ dostarttag
```

Now we will do all the detailed processing in the to-be-defined function **dostarttag**.

Before we do that: **xmlparser.parse** is now complete. The final listing looks like this:

```
xmlparser.parse = function (filename)
    local f = io.open(filename, "rb")
    local filedata = f:read("a*")
    f:close()

    local starttag = lt * lpeg.C(without(gt)^1) * gt / dostarttag
    local endtag   = lt * slash * lpeg.C(tagname) * gt
                    / xmlparser.endtag
    local data = lpeg.C(without(lt)^1) / dodata
    local pi = lpeg.P("<?")
              * lpeg.C(without(lpeg.P("?>"))^1) / xmlparser.pi
              * lpeg.P("?>")
    local comment = lpeg.P("<!--")
                  * lpeg.C(without(lpeg.P("-->"))^1)
                    / xmlparser.comment
                  * lpeg.P("-->")
    local parser = (pi + comment + endtag + starttag + data )^1
    lpeg.match(parser,filedata)
end
```

I am showing you the full listing because I want to remind you that the relative location of **starttag** in the line

```
    local parser = (pi + comment + endtag + starttag + data )^1
```

is of vital importance for proper functioning of this LPEG. I hope you can guess why, but if not: the problem is that we defined **starttag** such that the captured value is any sequence up to a **>** character:

```
local starttag = lt * lpeg.C(without(gt)^1) * gt
```

That definition would match processing instructions, comments, and end tags as well as the intended start tags.

It is possible to adjust the **starttag** definition to exclude input where the character immediately following the **<** is one of **/**, **!**, **?**. In this case, it is not necessary because

simply putting it after the other constructs works fine but if we *did* want to do so, it would look like this:

```
local q  = lpeg.P("?")
local ex = lpeg.P("!")
local starttag = lt * lpeg.C(without(gt+slash+q+ex)
                             * without(gt)^0)
                    * gt / dostarttag
```

In this, the capture is a character that is not in the set `>/?!`, optionally followed by any other character that is not `>`. Now **starttag** could be anywhere inside the **parser** definition. However, the LPEG would be a little bit more complex and thus run a bit slower.

Another option would have been to write a single **processtag** function and then instead of having four LPEGs for the four types of tags, just use a single one that would look like our original definition of **starttag**:

```
local tag = lt * lpeg.C(without(gt)^1) * gt / processtag
local data     = lpeg.C(without(lt)^1) / dodata
local parser   = (tag + data )^1
```

Yet another option would have been to add a sixth component to **parser**, by treating empty XML tags separately. Doing so would be quite simple, like this:

```
local etag = lt * lpeg.C(without(gt)^1 * slash) * gt / doemptytag
```

and then adding that just before **starttag** in the definition:

```
local parser = (pi + comment + endtag + etag + starttag + data)^1
```

I did not do that because empty XML tags can still have attributes so the **doemptytag** function would have to duplicate most of the functionality that is going to be needed in **dostarttag**. Also, I want to show a fairly simple way of handling both cases together inside the **dostarttag** function.

However, using separate LPEGs for start tags and empty tags is definitely an option, as is combining all the tag definitions into a single LPEG. It would even be possible to extend our **parser** definition to take care of the entity processing and attribute parsing inside the LPEG itself.

While you are still new to LPEG, large complicated LPEG objects can get confusing. As a beginner, it is much easier to pass off processing to a separate function, that can then define its own LPEG to deal with the details (as we did with the **dodata** function, and will do again in **dostarttag**).

So, let's talk about **dostarttag**. It needs to do two separate things: process the XML attributes (if they are present), and deal with empty XML tags.

First, I want to introduce another tiny helper function to take care of optional input:

```
local function maybe(p) return p^-1 end
```

This goes back to the table in the beginning of this article:

> '**patt^-n**: matches at most **n** repetitions of **patt**'

So the function **maybe** creates an LPEG that matches at most one appearance of its argument LPEG. We will use this function a couple of times but for now it is most helpful in recognizing empty XML tags. The overall structure of **dostarttag** looks like this:

```
local function dostarttag (tagdata)
    local emptytag = false
    local function isempty() emptytag = true end

    -- various other stuff to do here

    local tagparser = starttag * maybe(slash/isempty)
    lpeg.match(tagparser,tagdata)

    xmlparser.starttag(tag,attrs)
    if emptytag then
        xmlparser.endtag(tag)
    end
end
```

The **maybe(slash/isempty)** part recognises the empty tags. It sets a flag that is used to output the required **endtag** client event. The combined LPEG **tagparser** works by first finding the tag name and any attribute assignments. After that, a **/** can be left over at the end of the input string and that is the empty tag marker.

Note that **isempty** does not get any captures, as neither **slash** nor **maybe()** contain **lpeg.C** but it is still executed if there is a successful match. The **/** attaches the function to the right to the actual LPEG pattern on the left. Whenever the pattern succeeds, the function will be executed. The **isempty** needs to be inside of **maybe()**, otherwise it would be executed every time since the whole of **maybe()** is optional, thus it is always a successful match.

The definition of **starttag** itself is quite short:

```
local tag = ''
local attrs = {}
local function thetag (val) tag = val end
local function storeatt (att,val)
  attrs[att] = fixentities(val)
end
local starttag = lpeg.C(tagname)/thetag
                 * maybe(space * (attribute/storeatt)^1)
```

The **tagname** was already defined above. An XML start tag is a XML element name, optionally followed by a sequence of space-separated attribute assignments. The **space** is *inside* **maybe()** for the case where there is no attribute at all.

The definition of **attribute** will contain an optional trailing space, which serves as the separator between individual attributes as well as allow extra whitespace at the end of the attribute list, just before the closing **>**.

Each **attribute** will match one attribute but because of the **^**1, this LPEG will be matched repeatedly. The **storeatt** function will store the found attribute name and value into the **attrs** array.

Note that the call to **storeatt** has to be inside parentheses here, because we want the function to be called for each separate match. If it came after the **^**1, it would be called only once, with all the captures from all the **attribute** matches to that single function call.

Finally, we have to write the definition of **attribute**. Attribute definitions in XML as quite simple: it is a single 'word' that is the attributed name, followed by an equals sign, then a single- or double-quoted value string. The quote characters themselves inside the value have to be encoded as entities, so we do not have to worry about string escapes. Around the equals sign there can be optional whitespace.

The end result is (for this article) a fairly long definition, but quite an easy one to follow:

```
local attname   = without(space + eq)^1
local dattval   = without(qt)^1
local sattval   = without(sq)^1
local attribute = lpeg.C(attname)
                  * maybe(space) * eq * maybe(space)
                  * ((qt * lpeg.C(dattval) * qt) +
                     (sq * lpeg.C(sattval) * sq))
                  * maybe(space)
```

For **attname**, we could have reused **tagname** by adding the exclusion of **eq**. But the extra excluded characters in **tagname** (**gt** and **slash**) cannot happen inside attribute names anyway (because they have been intercepted by the enclosing LPEG matches), so we may as well create a new variable.

There are three calls to **lpeg.C** in the definition of **attribute**, but because single- and double-quoted attributes values are mutually exclusive options (signalled by the **+** operator), only two are ever called in a single match. So, the **storeatt** function always gets exactly two arguments.