Using ConTEXt with Databases

Accessing and typesetting information that is stored in databases is a common task. There are large-scale commercial solutions, and a number of database engines allow formatted output. In this article, I will show you one particular example: how I use ConTEXt MkIV to typeset material from a database. For my classes in Greek and Latin grammar, I have accumulated a large collection of exercises and examples from which I produce exercise sheets for my students. For a long time, I have relied on good old copy-and-paste to make new exercises and reuse some old material every year. But then I decided to do things in a more structured manner: I am in the process of putting all my examples into some sort of database from which the single exercise sheets will retrieve the exercises. This makes it easier to keep track of the material, make additions, and reuse elements in different ways without being too repetitive. In this article, I will demonstrate how ConTEXt can be applied to use such a database. There are two parts: in the first, you will see the new MkIV xml system in action; this new approach to processing xml from within ConTEXt makes it easy to access and manipulate parts of xml files. The second part will show a way to use sql databases as input for ConTEXt. I hope these examples can be useful for others who have similar needs.

The xml Database

The structure of our database is pretty simple: it has chapters covering single grammatical topics; every chapter has different examples. Every example has a unique identifier (expressed with an xml attribute `"id"`). There are two types of examples:

1. Normal examples have three elements: the “problem” (an English sentence or passage which is a translation of a Latin original), the “solution” (the Latin original), and the “origin” (the reference to the original, which is for my reference only and will not be typeset).
2. Some grammatical phenomena, however, can better be shown with Latin examples. In this case, we only have a Latin “problem” and an “origin.” These problems receive an identifier in the form of an xml attribute `type="latinonly"`.

Hence, a few examples from this database would look like this:

```
<examples>
  <chapter id="moods">
    <example id="deliberative1">
      <problem type="latinonly">
        quid ergo istius in iure dicundo libidinem et scelera demonstrem?
      </problem>
      <origin>
        Cicero, Verr. 2.39
      </origin>
    </example>
    <example id="indirect1">
      <problem>
        You do not see what he means.
```

```
      </problem>
      <solution>
        quid sentiat, non uidetis.
      </solution>
      <origin>
        Cicero, fin. 2.21
      </origin>
    </example>
    <example id="interdicere1">
      <problem>
        I have neither done it yet nor do I think it is forbidden to do it.
      </problem>
      <solution>
        id neque feci adhuc nec mihi tamen ne faciam interdictum puto.
      </solution>
      <origin>
        Cicero fin. 1.7
      </origin>
    </example>
  </chapter>
</examples>
```

The ConTEXt Environment

How can we make use of this xml database `examples.xml` in ConTEXt? We will use a ConTEXt environment to set up xml processing and format the output to our needs; this environment will be stored in a file `compositionxmlstyle.tex`. The first thing it does is define our environment:

```
\startenvironment compositionstyle

\stopenvironment
```

All the following lines go into this environment. We will now go through the TEX code step by step and see what it does.[1] We begin by simply loading our xml database:

```
\xmlloadonly{grammar}{examples.xml}{}
```

This simply makes the content of the database available to ConTEXt and it reserves the namespace `grammar` for this content.
We now have to process our database. There are two cases that we need to consider: the first is the "Problems" section. We want to be able to pick single problems, depending on their `id` attribute. Our first macro does just that: it extracts ("filters") a particular example:

```
\def\MyExample#1%
  {\xmlfilter{grammar}
   {/examples/chapter/example[@id=='#1']/command(xml:choose)}}
```

This macro is an instructive example of what the new MkIV xml mechanism can do.[2] As you see, it takes one argument, which it transfers to the command `\xmlfilter`. This command selects (or "filters") the content of our xml file (which is available under the name `grammar`). It traverses the structure of our xml file and picks the element `<example>` whose `id` corresponds to the argument of our macro; it then takes the content of the `<example>` element and transmits it to the command `xml:choose`.

Hence, this macro could be used in the form \MyExample{deliberative1} to pick the first example in our database.

```
\startxmlsetups xml:choose
  \doifelse {\xmlattribute{#1}{/problem}{type}} {latinonly}
    {\startitem[\xmlatt{#1}{id}]
      {\language[latin]\xmlstripped{#1}{problem}}
     \stopitem}
    {\startitem[\xmlatt{#1}{id}]
      \xmlstripped{#1}{problem}
     \stopitem}
\stopxmlsetups
```

When we "choose" our examples, we distinguish two cases: if the problem is of the "latinonly" type, it is a Latin phrase; otherwise, it is an English phrase. So we use the command \doifelse to distinguish between these two cases. This macro takes four arguments: the first two arguments are two strings that will be compared. If they are equal, the third argument will be executed; if they are not equal, the fourth. In our case, then:

- The command \doifelse looks at the attribute type of the sub-element problem of the current xml node (that's what \xmlattribute{#1}{/problem}{type} expands to).
- If the type attribute is equal to "latinonly", the first branch is executed: we produce an \item; its reference (in square brackets) is the id attribute (\startitem[\xmlatt{#1}{id}]). For the text of the \item, we switch to the Latin language (to get proper hyphenation). \xmlstripped takes the value of the xml element and strips leading and trailing spaces, so the text in the problem subelement is typeset as the content of the item.
- If we have a "normal" example, with a solution subelement, we do the same thing, but we do not switch to Latin (because the sentence is English).

So now our "problems" are wrapped up as ConTEXt items, ready to be processed in a \startitemize environment. Next, we look at the solutions. We write a similar macro that filters examples; this time, it passes their content to a different command:

```
\def\MySolution#1%
 {\xmlfilter{grammar}
  {/examples/chapter/example[@id=='#1']/command(xml:solution)}}
```

This macro works exactly like the \MyExample macro. Next, we define the command xml:solution. Again, we will use ConTEXt's conditional mechanism: The \doifnot macro only processes examples that do *not* have a problem subelement of the "latinonly" type (remember, only these have a solution):

```
\startxmlsetups xml:solution
  \doifnot {\xmlattribute{#1}{/problem}{type}} {latinonly}
    {\SolutionMargin{\in[\xmlatt{#1}{id}]}
      {\language[latin]\xmlstripped{#1}{solution}}
       \par\blank[line]}
\stopxmlsetups
```

Every problem was converted into an item which had its id attribute as a reference. The second example from our database would thus be processed by ConTEXt as

```
\startitem[indirect1]
  You do not see what he means.
\stopitem
```

Our `xml:solution` command now picks up this reference (`\in[\xmlatt{#1}{id}]` will expand to `\in[indirect1]`) and wraps it into a ConTEXt macro `\SolutionMargin` (which we will define shortly). It then switches to Latin, gets rid of unwanted spaces, and typesets the text of the `solution` subelement, followed by a paragraph and an empty line.
Now, we prepare the look of our exercise sheets. We want problems and solutions to look exactly like the same. In both cases, we want the numbers to appear in the margin, in bold. So we first define `\SolutionMargin` as a margintext which will be typeset in the left margin:

```
\defineinmargin [SolutionMargin] [left] [normal] [style=bold]
```

Then, we define an itemgroup for our problems which will also display its numbering in the margin:

```
\defineitemgroup[MyExamples]
\setupitemgroup[MyExamples][n,inmargin]
\setupitemgroup[MyExamples][style=bold]
```

Now, the last macro we have to define; this is the one that we will really use in our document. Since we are lazy and want to type as few words as possible when we prepare our exercise sheets, this macro will do all the work for us:

```
\def\MyExercises[#1]%
  {\startsubsection[title=Problems]
     \startMyExamples
       \processcommalist[#1] \MyExample \par
     \stopMyExamples
   \stopsubsection
   \startsubsection[title=Solutions]
     \processcommalist[#1] \MySolution \par
   \stopsubsection}
```

Do you see what this macro does? It takes a comma-separated list as argument. It then starts a subsection (with title “Problems”), and within this subsection, it starts our itemgroup `\MyExamples`. It then processes our comma list and hands every argument over to the macro `\MyExample`, which in turn retrieves the examples from our xml database. Since, as you remember, this macro calls the helper command `xml:choose`, this will take the content of the problem and pass it to a `\startitem`, with reference `id`. Then, the macro inserts a new subsection (with title “Solutions”), processes our comma list again and typesets all the solutions as we defined in our `xml:solution` command, viz., with the reference to the problem in the margin. This guarantees that the numbering of problems and solutions will be consistent.

The User Interface

After all this hard work, we can finally reap the benefits: when we prepare our exercise sheets, we will only have to do a minimum of typing: we include our environment, we give some structure in the form of sections, and we include a comma-separated list of the examples from the xml database that we want typeset. All the rest is done by the macros we defined. So our document will look like this

```
\environment compositionstyle

\starttext

\startsection[title={Moods}]
  \MyExercises[deliberative1,indirect1,interdicere1]
\stopsection

\stoptext
```

This will typeset exercise sheets, complete with examples and solutions. But wait: what if you first want to give out exercises to the students without the solutions? Of course, you could postprocess the resulting pdf file and pick only the pages with the problems, but that would not be very elegant. A better solution is to build this capability right into our environment. We will use ConTEXt modes. We modify our main macro:

```
\def\MyExercises[#1]%
  {\startsubsection[title=Problems]
     \startMyExamples
       \processcommalist[#1] \MyExample \par
     \stopMyExamples
   \stopsubsection
   \startmode[solutions]
   \startsubsection[title=Solutions]
     \processcommalist[#1] \MySolution \par
   \stopsubsection
   \stopmode}
```

Now, the part of our macro which typesets the solutions will only be executed if the mode `solutions` is set. You can either insert a line `\enablemode[solutions]` into your file, or, even easier, you can set the mode when you call ConTEXt from the command line. When you typeset your file and want to have the solutions as well, the command is: `context --mode=solutions`; if you don't enable this mode, only the problems will be typeset.

Further Elements

As an example of what else we can do, I'll show you how you can handle more xml elements. Alas, students are not as fluent in Latin as they used to be a mere 450 years ago, so they sometimes need a little bit of help. How can this be integrated into our documents? First, let us look at the xml side. In order to give subtle hints, we just invent a new xml element `<hint>`; here is an example:

```
<examples>
  <chapter id="moods">
    <example id="interdicere1">
      <problem>
        I have neither done it yet nor do I think it is forbidden
        <hint>interdicere</hint> to do it.
      </problem>
      <solution>
        id neque feci adhuc nec mihi tamen ne faciam interdictum puto.
      </solution>
      <origin>
        Cicero fin. 1.7
```

```
      </origin>
    </example>
  </chapter>
</examples>
```

We want these hints typeset in the text of the problems, between square brackets, in italics. How do we do this? First, we have to "grab" these elements from our loaded xml file and connect them with a setup command:

```
\xmlgrab{grammar}{hint}{xml:hint}
```

Then, we define our setup command:

```
\startxmlsetups xml:hint
      \dontleavehmode[{\language[latin]\em \xmlflush{#1}}]
\stopxmlsetups
```

Which will take care of the <hint> elements, apply Latin hyphenation, and typeset them as we want them.

Other Databases: sql

If you did not like the preceding paragraphs, I have something else to offer: you may have been disappointed that the title of this article mentions "databases," yet all it talks about is xml. What if you want to use a "real" database format such as sql? ConTEXt MkIV can also cope with sql databases – or, to be more precise: Lua can, and so can ConTEXt with the luaTEX engine. I do not have the knowledge to make an in-depth comparison of xml with sql. If you search the web, you will see that people have (sometimes strong) preferences for one or the other. One advantage of xml is that it's stored in simple text files, in a human readable form; you are thus sure that your database will be usable for a long time. sql, on the other hand, has its advantages when it comes to speed (though the speed of the lookups is relatively negligible compared to the time it takes to typeset the document, so unless your database is really huge, there should not be much of a difference). One rule of thumb seems to be that xml is better for data which has a strong hierarchical structure, whereas relational databases (such as sql) are better for large sets of weakly structured data. When you look at our database of grammatical exercises, you will see that there is not much hierarchical structure; what we really need is to retrieve single examples by their ids, and this is something that sql is very good at. It is thus easy to think of a way to represent our grammatical exercises as an sql database. The table will need five columns (I abbreviate the text to make this representation easier to read):

```
id            | type     | problem     | solution     | origin
------------- | -------- | ----------- | ------------ | -------------------
deliberative1 | latin    | quid...     |              | Cicero, Verr. 2.39
indirect1     | both     | You do...   | quid...      | Cicero, fin. 2.21
interdicere1  | both     | I have...   | id neque...  | Cicero, fin. 1.7
```

It is easy to see how this is (almost) identical to our xml file. We now have a column type to distinguish between sets with and without a solution. The other columns correspond exactly to our xml tags. (We lose the information about the grammatical "chapter" to which every example belongs; if we wanted to, we could add another column to our database carrying that information).
This is not the place to give an introduction to sql, so I will be very brief here: I chose the sqlite3 database management system because it is leightweight, open source,

available on many platforms, and does not rely on a server-client structure, hence it is well adapted for managing local databases.[3] `sqlite3` may already be available on your system, or else it can be installed quite easily. Creating such a database is easy. From the command line, we first create an empty database file:

```
sqlite3 grammar.db
SQLite version 3.7.3
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite>
```

We are now at the `sqlite` command prompt and can create our table:

```
CREATE TABLE examples (id TEXT PRIMARY KEY UNIQUE,
   ...> type TEXT,
   ...> problem TEXT,
   ...> solution TEXT,
   ...> origin TEXT);
```

This creates the structure (or "schema") for our table, and we can now insert our exercise problems; I give one example only:

```
INSERT INTO examples(id, type, problem, solution, origin)
   ...> VALUES('indirect1', 'both', 'You...', 'quid...', 'Cicero, fin. 2.21');
```

This will populate our table with our exercises, ready to be retrieved later.
How can we use an sql database in ConTEXt, then? Some years ago, Berend de Boer published a paper on this topic in the EuroTEX 2001 conference; of course, this applied to ConTEXt MkII.[4] He pointed out that it is fairly easy to insert xml tags or even ConTEXt commands into the output of an sql query. So one could massage the output and write it to a file, then call ConTEXt on that file. Berend proposed to do all this in a `perl` wrapper script. This would still be a viable route – but how much fun would that be? In MkIV, we can use a different approach: we can do the sql queries directly from within our document and typeset the results with ConTEXt. However, there are two caveats you will have to keep in mind when you read the remainder of this article:

1. I am not a database programmer by any means, I just impersonate one for this article. The code I will show you here does work for me, but it may be quite naive or unsophisticated – you are welcome to improve it!
2. Unfortunately, what I have written in the first paragraph of this section is not quite literally true: Lua can indeed deal with sql, but it needs additional modules to do so. Unfortunately, I found the situation a bit confusing: there are (at least) three different modules that allow Lua to work with sqlite databases.[5] Even after doing some research on the web, I could not quite figure out in which relation these modules are – they are quite similar in their basic approach, but differ in many aspects of the user interface.[6] For this article, I use the Lua module `LUASQLite3`,[7] which appears to be the only one which is still actively maintained. In another article in this issue, Taco Hoekwater explains how to install a Lua module so that luaTEX and ConTEXt can actually use it.

With all this understood, let us set our specifications for what we want to achieve, then. We want to keep exactly the same user interface in our ConTEXt file as in the first part of this article when we were dealing with xml; i.e., we still want to use our

macro \MyExercises[] with a comma list of examples; and we want to be able to use modes to have our solutions typeset or not. Our aim is to produce a universal macro that can be driven either by an sql database or by an xml file, without the user having to worry about it.

So let us roll up our sleeves: our database grammar.db is in place; its table examples is populated with our exercises; our luatex binary is able to find and use the lsqlite3 module. What would our new environment look like? Much of what we will do now will be done in Lua, so we begin by writing our Lua code. I find it convenient to write and test my Lua code first and wrap it into the proper \startluacode \stopluacode environment later, but this is just a habit.

We could, of course, reuse some of the code we have written for handling the xml file, especially the part where we used the neat \processcommalist macro, but since we will be writing a Lua function anyway, I found it interesting to see how much of this could be done in Lua. As you will see, it is possible to write the cimplet set of processing and typesetting commands in Lua. This may sometimes appear a bit convoluted, but it may inspire you when you want to write your own Lua functions, so here we go: our Lua function (which will later be wrapped into a \ctxlua macro) will take as its argument a comma-separated list of values. So the first thing we have to do is split this list into its elements and make a table out of them; we use the lpeg library for this.[8] In the following code, the variable keywordlist designates what will be the user input when we define our ConTEXt macro.

```
userdata = userdata or { }
userdata.sql = userdata.sql or { }
userdata.sql.sep = lpeg.P(",")
userdata.sql.mywords = lpeg.C(( 1 - userdata.sql.sep )^0)
userdata.sql.p = lpeg.Ct(userdata.sql.mywords *
                 (userdata.sql.sep * userdata.sql.mywords)^0)
userdata.sql.mytable = lpeg.match(userdata.sql.p, keywordlist)
```

This creates a Lua table userdata.sql.mytable with all the keywords in it. We will later use this table to retrieve the single problems from our database. But let us first look at the way we will be accessing the database itself.

The Lua function proper will query the database for entries whose id corresponds to this element. Here is how this can be achieved:

```
require("lsqlite3")

userdata.sql.mygrammar = assert (sqlite3.open("grammar.db"))

function userdata.sql.getproblem(myid)
  userdata.sql.myquery =
    userdata.sql.mygrammar:prepare("SELECT problem FROM examples WHERE id = ?")
  userdata.sql.myquery:bind_values(myid)
  userdata.sql.myproblem  = userdata.sql.myquery:get_value(0)
  userdata.sql.myquery:step()
  userdata.sql.myquery:finalize()
end
```

Let us have a brief look at this code. This is the part where Lua interacts with our database. The first thing we have to do is load ("require") the lsqlite3 module. We use it to open our database and give a symbolic name to the resulting Lua structure. We then run an sql SELECT query on the table examples in this database. This query creates an object, to which we again assign a handle, userdata.sql.myquery. The interesting part here is the end of the query: in WHERE id = ?, the question mark

is a placeholder which we then "bind" to the argument of our Lua function, `myid`. Our query selects the column `problem` from the database which is captured in the function call `userdata.sql.myquery:get_value(0)` (if we wanted to retrieve `n` more columns, these would be captured as `userdata.sql.myquery:get_value(0+n)`). We assign a Lua variable to this result. The function call `userdata.sql.myquery:step()` will actually apply our query to the next row of the database; the query is closed with `userdata.sql.myquery:finalize()`.
So all we have to do now is write a loop which will take the single elements of our `userdata.sql.mytable`, make sure to get rid of all whitespace which users may put into this comma list, pass the single values on to the `userdata.sql.myquery` function, and then do something with the results we receive. Since we want the results to be the same as with the xml example, we reuse the setups for our item lists and our margin numbers:

```
\defineinmargin [SolutionMargin] [left] [normal] [style=bold]

\defineitemgroup[MyExamples]
\setupitemgroup[MyExamples][n,inmargin]
\setupitemgroup[MyExamples][style=bold]
```

And this is how we will use these definitions in our Lua loop:

```
context.startsubsection( { "title=Problems" } )
context.startMyExamples()

for k, myid in ipairs(userdata.sql.mytable) do
  myid = myid:gsub(" ", "")
  userdata.sql.myquery =
    userdata.sql.mygrammar:prepare("SELECT problem, type
                              FROM examples WHERE id = ?")
  userdata.sql.myquery:bind_values(myid)
  userdata.sql.myquery:step()
  userdata.sql.myproblem = userdata.sql.myquery:get_value(0)
  userdata.sql.mytype = userdata.sql.myquery:get_value(1)
  userdata.sql.myquery:finalize()
    context.startitem( { myid } )
      if userdata.sql.mytype == "latin" then
        context.bgroup()
        context.language( { "latin" } )
        context.delayed(userdata.sql.myproblem)
        context.egroup()
      else
        context(myproblem)
      end
    context.stopitem()
end
context.stopMyExamples()
context.stopsubsection()
```

What you see here is ConTEXt code written in Lua. Every ConTEXt command has a corresponding Lua equivalent. If you define an environment `\MyExamples`, the Lua function call `context.startMyExamples()` is equivalent to `\startMyExamples`.[9] As I said before, we could have done most of this in ConTEXt itself; I just wanted to demonstrate this Lua interface here.

But there is more! Remember, we also wanted to typeset the solutions for problems that were of `type latinonly` if the mode `solutions` was set. Here is how we can do this in Lua:

```
if tex.modes["solutions"] then
  context.startsubsection( { "title=Solutions" } )
  for k, myid in ipairs(userdata.sql.mytable) do
    myid = myid:gsub(" ", "")
    local userdata.sql.myquery = userdata.sql.mygrammar:prepare
            ("SELECT type, solution FROM examples WHERE id = ?")
    userdata.sql.myquery:bind_values(myid)
    userdata.sql.myquery:step()
    userdata.sql.mytype = userdata.sql.myquery:get_value(0)
    userdata.sql.mysolution = userdata.sql.myquery:get_value(1)
    userdata.sql.myquery:finalize()
    if userdata.sql.mytype == "both" then
      context.SolutionMargin(context.delayed["in"]( { myid } ) )
      context.bgroup()
      context.language( { "latin" }, context.delayed(mysolution) )
      context.egroup()
      context.par()
    end
  end
  context.stopsubsection()
end
```

As you can see, we have to query the database a second time, to retrieve the solutions. This time, we also need to retrieve the `type` column since only database entries with `type both` do, in fact, have a solution, and we need to test this (otherwise, Lua will complain because the instruction `context(userdata.sql.mysolution)` may result in an empty argument). You will find it easy to recognize the other elements which we have already done in the first part: we test whether the mode `solutions` is enabled; if it is, we further test whether the `type` of the entry is `both`; if it is, we typeset it, with its `id` as a reference to the item in the problem list.
So all we need to do now is wrap the entire Lua code in the proper environment (I give just the beginning and the end) and define the command that we will use in our file:

```
\startluacode
require("lsqlite3")
userdata.sql.sep = lpeg.P(",")
function userdata.sql.getexample(keywordlist)
...
end
\stopluacode
\def\MyExercises[#1]%
  {\ctxlua{userdata.sql.getexample("#1")}}
```

As you see, in our sql environment, the macro `\MyExercises` passes its argument over to our Lua function `getexample`, which will in turn split it into its single keywords, query the database for them, and finally typeset the corresponding problems and solutions. So we have achieved exactly what we wished: we have defined the same macro which will now retrieve our exercises from an sql database!

What about our special <hint> element? If you remember, we wanted these hints typeset within brackets, and in italics. How can we integrate this into our sql approach? One solution would be to write the ConTEXt code which you want evaluated directly into the entry in the database, so the problem column of our example would look like this:

```
... it is forbidden [{\language[latin]\em interdicere}] to do it.
```

If you are certain that you will never use your database with any other (necessarily inferior) tools than ConTEXt, this would be a possible way, but it is not very elegant. Better to keep the database as generic as possible and massage the data at the ConTEXt end. So we have to think of a delimiter for our hints – this must be a pair of characters that you will not use in any other way. In our case, square brackets are used for nothing else but to include such hints, so our database simply contains:

```
... it is forbidden [interdicere] to do it.
```

When we retrieve the problems in our Lua code, we simply replace these brackets with the code:

```
function userdata.sql.debracket(s)
   p = string.sub(s,2,-2)
   return p
end

userdata.sql.myproblem = userdata.sql.myproblem:gsub
  ("(%b[])", function(t) return "[{\\language[latin]\em"
   .. userdata.sql.debrac(t) .. "}]"; end)
```

This operation on the string with Lua's gsub command will simply replace all strings within balanced brackets (such as [interdicere]) in the output of our query with [{\language[latin]\em interdicere}].

Conclusion

The actual output of our exercise sheets doesn't look very exciting yet (actually, “Problems” and “Solutions” will be typeset on two different pages, but here I have indicated the page break by a simple line).

Moods

Problems

1 quid ergo istius in iure dicundo libidinem et scelera demonstrem?
2 You do not see what he means.
3 I have neither done it yet nor do I think it is forbidden [*interdicere*] to do it.

Solutions

2 quid sentiat, non uidetis.
3 id neque feci adhuc nec mihi tamen ne faciam interdictum puto.

But it is easy to add bells and whistles, color, different fonts and sizes, etc. It's all a matter of adapting settings in your environment. The example I have shown here may be a bit specialized, but it should allow you to appreciate the simplicity of the underlying mechanism.

Footnotes

1. As always, I would not have been able to figure all this out myself. I gratefully acknowledge the help of the ConTEXt community on the mailing list; in particular, Aditya Mahajan and Peter Münster have provided valuable help. And, as always, none of this would have been possible without Hans Hagen's kind support.
2. If you are curious and want to know more about xml in ConTEXt MkIV, you should have a look at the manual which can be downloaded at http://www.pragma-ade.com/general/manuals/xml-mkiv.pdf.
3. For more information, point your browser at http://sqlite.org/.
4. The paper is available at http://www.ntg.nl/eurotex/deboer.pdf.
5. See the somewhat terse wiki page at http://lua-users.org/wiki/LuaSqlite.
6. It is somewhat reassuring to see that other users feel confused, too; see the questions at http://lua-users.org/lists/lua-l/2009-03/msg00405.html.
7. See http://luaforge.net/projects/luasqlite/.
8. The following code is adapted from the `lpeg` website at http://www.inf.puc-rio.br/~roberto/lpeg/lpeg.html#ex.
9. For more information, see Hans Hagen's article "Typesetting in Lua using LuaTEX" in the previous issue of *MAPS* and the manual at http://www.pragma-ade.com/general/manuals/cld-mkiv.pdf.

Thomas A. Schmitz