Using TEXLua for track plan graphics

Taco Hoekwater

TEXLua, combined with some of the Lua library files from ConTEXt, can easily be used to do parsing of almost any file format. I plan on using that approach to generate graphics from my model railroad track plan that is itself designed in XtrackCAD. The lpeg library and some helpers are used to parse the file format and generate MetaPost source that will be converted into png images.

Introduction

For much of my twenties and thirties, I spent way over half of each calendar day behind a computer screen. Almost fifty now, I am finally loosing interest a little, and I went back to a hobby from my teens: model trains.

Strangely enough, there is a lot of staring at computer screens involved with this rediscovered hobby. In the last thirty years, computers have become quite integrated with model railroading: there are programs to help design your layout, programs to control turnouts, programs that actually drive the train engines, programs to monitor where those trains are on the layout, and then of course there are programs to actually control those programs.

So now I can have the best of both worlds: playing with all these programs is a lot of fun, however when (occasionally) that gets boring, I can go off and actually play with toy trains!

Since I am starting from scratch (none of my old stuff was even rescuable after being taken through half a dozen moves, stored in crappy boxes on leaky attics, and generally having been neglected for three decades) the first task was to come up with a track plan. The program I use for that is called XtrackCAD, and I am really happy with it. Not that I had too much choice, working primarily on a Mac does limit the options quite a bit.

Still, XtrackCAD has a number of advantages over the mostly commercial Windows software options:

- It is open source software (mercurial on sourceforge), with prebuilt distributions for Windows, Mac, and Linux, with an active development team.
- It supports lots of track manufacturers with a database of measured parts (including mine!) and adding new part definitions is possible.
- It can work on multiple layers.
- It allows the addition of meta-information like block and turnout identifiers.
- It can create printouts in any scale, including at full size.

- It is a CAD-style program, so measurements can be exact.
- Its save files are ascii-based, meaning post-processing is easy.

Of course there are many other things worth mentioning when comparing XtrackCAD to other programs, but that is the list of features that I need the most. If you are interested yourself, have a look at http://www.xtrkcad.org. Below is a screenshot of the bottom layer of my current track plan in figure 1.

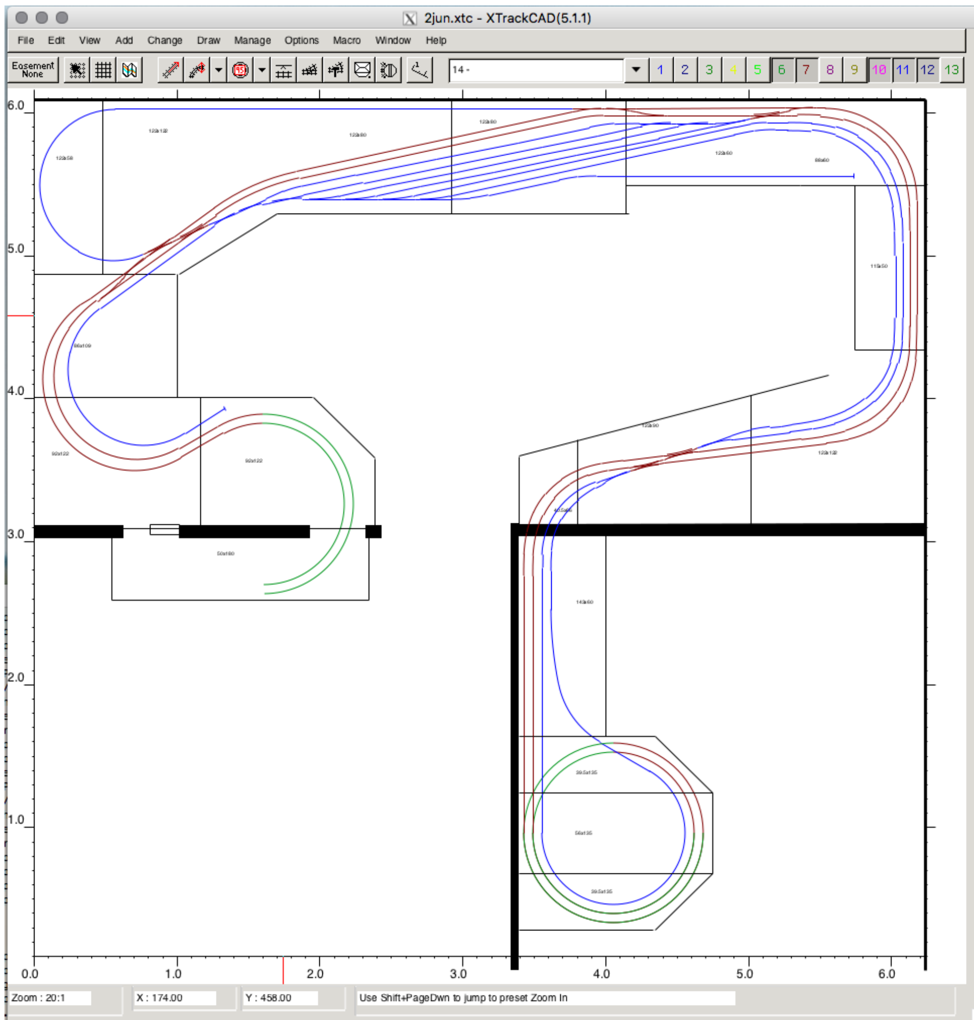

Figure 1. XtrackCAD in action

Even with all those features, there are a few things I wish it could do that it cannot. Most importantly, I want to document what I set up for the computer controls of the layout. For example, groups of track segments are combined into logical blocks for the computer control software to run trains.

This is separate from and not indicated by the colors on the screen, those colors are just visual aids for me while editing. The brown color is the ‘main line’, The blue

parts are staging and storage tracks, the green bits are the start of the helixes that go to the upper level.

On the partial layout you see in the screenshot, there are actually about 20 logical blocks, with most of those themselves divided into three sub-blocks. It is possible to enter that logical block information using a popup dialog in XtrackCAD, but there is no way to create a nice visual overview. I could technically put each in a separate layer with a slightly different color, but with so many colors that would quickly become confusing. Besides, that would still not display the logical name of the block, nor the identifiers of the sensors attached to it.

Alternatively, I could produce a multi-page printout and then do the labelling manually. Or export a PDF and add notes on that. But the downside of those is that when something changes on the layout, it is going to be hard to keep the manual documentation up-to-date. And with this being version 185, future changes seem highly likely.

Other options: I could nag the developers to create a nicer interface. Or I could jump in myself and start programming on XtrackCAD myself. That latter option would take quite a bit of studying though, and with the former option the final result would perhaps not be exactly what I want.

Besides, I asked myself: will the output ever be as good looking as I am sure it could become using LuaTEX and MetaPost? Surely not!

So, I went off to investigate the XtrackCAD save file format in detail. It looked rather doable to write an lpeg pattern matcher to convert its content to a Lua table. That can then be used to generate nice MetaPost graphics with nicely typeset LuaTEX captions for the label information.

At this point in time, I have not gotten very far. That is: the file format parser is complete and it can roundtrip the input so I can verify it is fine. And there are some rudiments of a MetaPost output stage. That's about it. Life got in the way, but at least now I am certain I will have something to talk about at next year's meeting.

XtrackCAD File format

Like I wrote earlier, the file format is text based. The general structure is that an uppercase word at the start of a line starts a record. Some records are a single line, other records are multi-line. In either case, the first line is a command word, followed by any arguments. In the multi-line record format, all subsequent lines are indented. Each of those lines is also a single command (of a type that only happens within record content), and the record ends with the command `END`. Lines that start with a hash mark are comments, and empty lines between records are ignored.

Here is the start of the file, containing some global information fields:

```
#XTrackCAD Version: 5.1.0, Date: Sat Jun  2 11:57:26 2018
VERSION 10 3.0.0
```

```
TITLE1 MGB
TITLE2 Bovenkamer
MAPSCALE 64
ROOMSIZE 245.669291 x 239.763780
SCALE HO
```

There are two versions here. The `Version` is the program version, the `VERSION` is the save file version. The `Date` is the timestamp of the save file. `MAPSCALE` is the scale for the popup navigation map. `ROOMSIZE` is interesting only because it is in inches: in fact all dimensions in the save file are in inches, even though the interface can be set to metric.

Next come the layer definitions:

```
LAYERS 0 1 0 1 255 0 0 0 0 "top tables"
LAYERS 1 0 0 1 128 0 0 0 0 "ns"
LAYERS 2 0 0 1 32768 0 0 0 0 "sgb"
LAYERS 3 0 0 1 14941952 0 0 0 0 "detection top"
LAYERS 4 0 0 1 65280 0 0 0 0 "helix1"
LAYERS 5 1 0 1 366623 0 0 0 0 "helix2"
LAYERS 6 1 0 1 8388608 0 0 0 0 "bottom"
LAYERS 7 0 0 1 8388736 0 0 0 0 "detection bottom"
LAYERS 8 0 0 1 8421376 0 0 0 0 "helix3"
LAYERS 9 1 0 1 16711935 0 0 0 0 "staging2"
LAYERS 10 1 0 1 255 0 0 0 0 "staging"
LAYERS 11 0 0 1 128 0 0 0 0 "bottom tables"
LAYERS 12 0 0 1 32768 0 0 0 0 "staging3"
LAYERS 13 0 0 1 16776960 0 0 0 0 ""
LAYERS 14 0 0 1 65280 0 0 0 0 ""
LAYERS CURRENT 0
```

What makes this interesting is that it highlights a problem of nearly all free software: the documentation does not quite match up with reality. In the case of XtrackCAD, there is a wiki page to document the file format. Here is what it has to say about the `LAYERS` command:

```
LAYERS<sp>Layer-num<sp>visible<sp>0<sp>unlocked<sp>"Layer-Name"
```

```
Note - This is repeated for each defined layer
```

That would mean the save file should look like this according to the documentation:

```
LAYERS 0 1 0 1 "top tables"
```

But there are five extra arguments. So what are those? Snooping through the C source files reveals that the first of those is the layer color. The wiki explains that all colors are coded as (Red*65536+Green*256+Blue). So, layer `0` is 100% blue. The other four values are hardcoded zeroes, probably for a planned extension.

In my parser, I have not always been so diligent looking through the C source. Mostly, I combined what I saw on the screen with what I read on the wiki, and if that seemed to match up to command arguments, I just went with it and ignored any extra or missing arguments compared to the wiki specification.

After the layer definitions, the actual layout objects start. The order in which the commands are listed in the save file seems based on creation order, but that is not too important. What is relevant is that all the layout objects have an index number, and these are numbered consecutively.

Let's look at the first one in my file to illustrate some other points (by the way, notice that layer indices start at zero but object indices at one?)

```
DRAW 1 0 0 0 0 71.336544 -27.078188 0 336.079608
       F4 0 0.000000 4 0
              166.043307 107.972441 0
              222.736220 107.972441 0
              222.736220 99.212598 0
              166.043307 99.212598 0
       END
```

Here is what the wiki says:

```
DRAW<sp>index<sp>layer<sp>0<sp>0<sp>0<sp>start-x<sp>start-y<sp>angle
A set of line segments
END
```

Parsing this (and with a bit of correction), we should get:

argument	**value**	
index	1	
layer	0	
start-x	71.336544	(181.19cm right from the bottom left of the room)
start-y	-27.078188	(68.78cm below the bottom left of the room)
angle	336.079608	(0 degrees is 'up', and direction is clockwise, so a little after eleven o'clock)
segments	F4	

That extra zero between `start-x` and `angle` is ignored in the current XtrackCAD, and probably intended for a future 3D extension.

The `F4` command indicates a filled polygon. The header line says that the color is `0` (black), the line width is `0.000000`, the number of defined points is `4`, and the subtype is `0` (which stands for 'freeform').

Then the four points are defined. The third value is always zero right now, and is again intended for a future extension for elevation. The four points describe a rectangular box, which is then rotated 336.079608 degrees around the

(71.336544, −27.078188) point. Sounds a little odd? I thought so too, so I decided to check manually.

Ignoring the imperial to metric conversion, the MetaPost input would look like this (with the angle negated because XtrackCAD and MetaPost do not agree about clock direction):

```
p := (166.043307,107.972441)--
       (222.736220,107.972441)--
       (222.736220,99.212598)--
       (166.043307,99.212598)--cycle;
fill p rotatedaround((71.336544,-27.078188), -336.079608);
```

And indeed this reproduces what I see on my XtrackCAD screen. Why the polygon is so oddly defined as rotating around a point that is literally meters away, I have no idea… perhaps it is a side-effect of how I created or rotated the polygon in the GUI?

One last note to make about the `F4` command: this is actually called an `F` command with internal version `4`. This is how save format updates are handled. In the `F3` version, there was no subtype, and in the original `F` version there were no elevations in the point definition either.

The other commands in the save file follow the same pattern. Some commands are simpler, others are more complex. Two quick examples with going into the specific details of the arguments will end this section. First, this example is of a simple straight bit of track:

```
STRAIGHT 11 1 0 0 0 HO 2
       T4 12 165.221427 72.459726 141.000000 0 0.0 0.0 0.0 0.0 0 0 0 0.000000
       T4 16 162.155016 76.246429 321.000000 0 0.0 0.0 0.0 0.0 0 0 0 0.000000
       END
```

This says that this bit of track is connected to the object with index 12 at point (165.221427 72.459726) an angle 141 degrees, and also to the object with index 16 at (162.155016, 76.246429) on the opposite side (321 minus 141 equals 180, as it should be for a straight bit of track).

Worth noting is that the main object of the record does not have a direction angle. Instead, the angles of the connecting tracks are specified. This is true for all records that list track connections.

Second, here is the record that defines a turnout:

```
TURNOUT  6 1 0 0 0 HO 2 156.870079 237.051181 0 0.000000 "Peco RH Medium Turnout,
                                                        Electrofrog SL-E195"
       T4 393 156.870079 237.051181 270.000000 0 0.0 0.0 0.0 0.0 0 0 0 0.000000
       T4 17 165.492126 237.051181 90.000000 0 0.0 0.0 0.0 0.0 0 0 0 0.000000
       T4 5 165.492126 236.051181 102.000000 0 0.0 0.0 0.0 0.0 0 0 0 0.000000
       D 0.000000 0.000000
```

```
        P "Normal" 1 2
        P "Reverse" 1 3 4
        S 0 0.000000 0.000000 0.000000 0.648600 0.000000
        S 0 0.000000 0.648600 0.000000 8.622047 0.000000
        C 0 0.000000 31.101230 0.648476 -31.101230 0.000076 12.000152
        S 0 0.000000 7.114985 -0.679653 8.622047 -1.000000
        END
```

A quick legend will help understand this better:

- `T` Stands for 'Track connection'
- `D` Offset for the description label in the UI
- `P` Paths a train can take. The numbers reference the next four lines and indicate the internal routing
- `S` Straight internal section of the turnout (these use a local coordinate system)
- `C` Curved internal section of the turnout

A MTXRUN script

At this stage, I do not really need a command line interface, because there is very little functionality. The only actions I can perform are (attempt to) parse an XtrackCAD file, attempt to regenerate the file from the parse tree, and output rudimentary MetaPost code for testing the interpretation of the parse tree.

But I envisage lots of functionality appearing in the future, so it makes sense to create an mtxrun script right away. As a bonus, this allows the various `.lua` code files to live in my context tree instead of in a local or hardcoded directory.

```
xtrkcad         | XtrackCAD file support
xtrkcad         |
xtrkcad         | --xtc                  output a new xtc file (--output)
xtrkcad         | --metapost             output metapost input (--output)
xtrkcad         | --output=file          write to file instead of stdout
xtrkcad         | --test                 test the xtc parsing by roundtripping
xtrkcad         |
xtrkcad         |
xtrkcad         | Examples
xtrkcad         |
xtrkcad         | mtxrun --script xtrkcad --test file.xtc
xtrkcad         | mtxrun --script xtrkcad --metapost --output=file.mp file.xtc
```

With the `mtx-xtrkcad.lua`, `xtc-parser.lua` and `xtc-metapost.lua` stored in the folder `texmf-project/scripts/context/lua/third/xtrkcad`.

The `--test` option only returns a message indicating 'ok' versus 'not ok', without line numbers. I find getting lpeg to report problems in a usable format quite hard and am still looking for a way to generate nice error messages. For now, the way to check where the script goes wrong is by making it create new `.xtc` file, and then running the `diff` command manually.

The LPEG parser

From the lpeg website at http://www.inf.puc-rio.br/~roberto/lpeg/ 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 important word 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 specialized grammar that can interpret your input. lpeg offers a mix of overloaded Lua operators and special functions that allow you to break up the input into grammatically sensible parts (that you can then do whatever you want with).

There are quite a number of functions and operators defined by the lpeg module. Some are really basic, others are quite specialized. To keep my parser simple, I have used the simplest lpeg features that I could. This is mostly future-proofing myself: my memory is not the best, so the simpler the code, the less I will have to relearn when I inevitably will need to update the parser a few years from now.

Here are all the functional parts of lpeg that I am actually using:

Operator	Description
`lpeg.P(n)`	Matches exactly n characters
`lpeg.P(string)`	Matches string literally
`lpeg.S(string)`	Matches any character in string (Set)
`lpeg.R("xy")`	Matches any character between x and y (Range)
`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)
`lpeg.match(patt,input)`	match a pattern against some input
`lpeg.C(patt)`	the match for patt plus all captures made by patt
`lpeg.Ct(patt)`	a table with all captures from patt
`patt / function`	the returns of function applied to the captures of patt

I will introduce these commands better where they are used.

The lpeg parser module starts with a bunch of `local` definitions, as is typical of Lua code inside ConTEXt:

```
local match = lpeg.match
local P = lpeg.P
```

```
local S = lpeg.S
local R = lpeg.R
local C = lpeg.C
local Ct = lpeg.Ct
```

Following that, there are some basic definitions of parser units. When writing a parser, I find it works best to work from both ends: from the bottom up to find the smallest objects that still make sense as a parsed unit (strings, numbers, etc.) as well as from the top down to find the global structure of the input in terms of records.

There are two tiny helper functions that make it easier for me to remember the lpeg syntax:

```
local function maybe(p) return p^-1 end
local function without(p) return (P(1)-p) end
```

The `p^-1` syntax means: match at *most* one occurrence of the pattern `p`. But I find that not very readable, so I use `maybe(p)`. A similar situation exists for `P(1)-p`: `P(1)` just means: match any character. The minus operator modifies that so that the expression becomes: match any character that does not include whatever the pattern `p` is.

Next are some actual lpeg definitions:

```
local space       = S(" \t")^1
local emptyline   = P("\n") * maybe(space)
local nl          = P("\n") * maybe(space)
local quote       = P('"')
local dot         = P(".")
local digits      = R("09")^1
local endfile     = P("END")
local endcontent  = P("\tEND")
```

So a `space` is either a space or a tab character. The `emptyline` and `nl` consist of a line feed followed by an optional `space`. These two are different names for the same content, just to make the rest of the parser easier to read. `quote` is used to find strings; `dot` and `digits` for floating point numbers.

The `endfile` and `endcontent` are used for discovering the end of the input and the end of the current record.

The basic building blocks of `.xtc` files are integers, floats and strings:

```
local integer     = C(digits)
local float       = C(maybe(P("-")) * digits * maybe(dot*digits))
local string      = quote * C(without(quote)^0) * quote
```

This is the point where the `lpeg.C()` function becomes useful, as whenever we find an `integer`, `float`, or `string`, we want to capture them for further processing.

This is also the point where intimate knowledge of the file format comes in handy: I am certain, for example, that floating numbers without leading zero (like e.g. `.01`) never happen, nor do floats in scientific notation (like e.g. `1E-2`). If that was not the case, then the definition of `float` would have to be a bit more complex. Similarly, I know that I never use " within my labels in XtrackCAD. And since they cannot happen in the input, I do not have to worry about string quoting issues either.

After this, it starts to get a bit more functional:

```
local rest        = without(nl)^0
local content     = C(without(endcontent)^0)
local scale       = C(P("HO") + P("N"))
local version     = C(digits * dot * digits * dot * digits)
```

The `rest` is an easy way to grab the rest of a line. The `content` is a way to capture all of the content of a complex command. `scale` and `version` are low-level parser objects. Both of these will be used to scan the initial portion of the `.xtc` save file.

But before we get there, I have another block of convenience definitions

```
local function Sp(a)
  if a then return a * space else return space end
end
local oneF   = float
local twoF   = Sp(oneF) * oneF
local threeF = Sp(twoF) * oneF
local fourF  = Sp(twoF) * twoF
local oneI   = integer
local twoI   = Sp(oneI) * oneI
local threeI = Sp(twoI) * oneI
local fourI  = Sp(twoI) * twoI
```

These have no other function than to save typing and thus make the rest of the lpeg parser more readable. One other line of lua code worth mentioning now is this one:

```
xtc = {}
```

The variable `xtc` is a table that will hold all of our interpreting functions. In the code we will say for example:

```
C(rest / xtc.do_comment)
```

and this will pass the `rest` as argument to the function `xtc.do_comment`. The function itself is boring, it just saves the comment to the layout. Another case

of ‘know your input’ happens here, because I know there is always only a single comment line in the input, so a single variable in the parsed data structure is enough.

```
local layout = {}
xtc.do_comment = function (a)
    layout.comment = a
end
```

The ‘command record’ parser grammar lines for the initial portion of the saved file look like this:

```
local xtc_comment  = P('#') * (rest/xtc.do_comment) * nl
local xtc_version  = P('VERSION')  * space
                     * (Sp(oneI) * version / xtc.do_version) * nl
local xtc_title1   = P('TITLE1')   * space
                     * (rest/xtc.do_title1) * nl
local xtc_title2   = P('TITLE2')   * space
                     * (rest/xtc.do_title2) * nl
local xtc_mapscale = P('MAPSCALE') * space
                     * (oneI/xtc.do_mapscale) * nl
local xtc_roomsize = P('ROOMSIZE') * space
                     * (Sp(oneF) * Sp(P('x')) * oneF
                        / xtc.do_roomsize) * nl
local xtc_scale    = P('SCALE')    * space
                     * (scale/xtc.do_scale) * nl
```

A few notes on that. First, I could have written the second line like this as well:

```
local xtc_version  = 'VERSION '
            * (integer * ' ' * version / xtc.do_version) * '\n'
```

Because whenever the lpeg operators encounter a bare string or number, they will assume you actually meant to add a `P()` around that bare string or number. Sometimes this makes your grammar easier to read. But like all automatic conversions, it can also become confusing in odd cases. For example, this would not work unless extra braces are added around the string concatenation:

```
local xtc_version  = 'VERSION' .. ' '
            * (integer * ' ' * version / xtc.do_version) * '\n'
```

So I find it better to always wrap all strings and numbers in `P()`, just in case.

You can see from the actual code that I prefer to use actual lpeg definitions for everything expect keywords. This makes it easier to change something if (or rather when) the file format changes in the future. Currently, the `.xtc` format is very clear

about when it uses spaces versus tabs. But with active development being done to the program, that is one of those things that can quickly change on the whim of a XtrackCAD developer.

Final note on this part: the location of the `/ xtc.doversion` matters! When you add a function following a / operator, it will receive all the matches in the (sub)pattern, in the order in which they were found. What I want is for `do_version` to get the `integer` as its first argument, and the `version` as its second argument. Both of those have their own embedded `C()` and these are the two in this sub-pattern, so for my actual input it works out fine either way.

But imagine for a moment that there is a third `C` around these two matches, so if the input looked like the following example:

```
... C(Sp(oneI) * version) / xtc.do_version * nl
```

In that case, the function would receive three arguments. Argument one would be the input segment containing both of the matches I actually want, and arguments two and three would be those matches separately.

Similarly, if there was another match happening earlier in the line, like this:

```
local xtc_version  = C('VERSION') * space
                     * (Sp(oneI) * version) / xtc.do_version * nl
```

the first argument would be the `VERSION` match, and arguments two and three the ones I actually want.

Notice that the set of `()` does not actually affect the number of matches, the sole point of that extra set of `()` is to limit the scope of the / operator. Any matches outside of the enclosing `()` pair will not be fed into `do_version`. In this case that is not relevant at all, but it can make quite a difference in more complicated (sub-) patterns.

Let’s look at a somewhat more complicated case, like `DRAW`:

```
local xtc_head = Sp(oneI) * Sp(oneI) * Sp(oneI) * Sp(oneI) * oneI

local xtc_draw = P('DRAW') * space * (Sp(xtc_head) * Sp(twoF)
                 * Sp(oneI) * oneF * nl * content  / xtc.do_draw)
                 * endcontent * nl
```

The `do_draw` function will get a much longer list of arguments. Its definition looks like this:

```
xtc.do_draw = function (index,layer,a,b,c,startx,starty,d,
                        startangle,content)
    local t = {
```

```
          index  = tonumber(index),
          type   = "draw",
          layer  = tonumber(layer),
          startx = startx,
          starty = starty,
          angle  = startangle }
      handle_content(t,content)
      layout.track[tonumber(index)] = t
      layout.lasttrack = tonumber(index)
  end
```

The arguments `a`, `b`, `c` and `d` are the hardcoded zeroes explained earlier. They are parsed since you cannot ignore them without extra work, but they are not stored. Worth noting:

1. All pattern matches are always strings. If you need numbers, you have to convert them yourself.
2. The results are stored in the `layout` data structure as a type of `track`. That may seem odd, but since `index` values are shared between track items and drawing objects like this, it makes sense to keep them together so that when it comes time to do processing of the data, `ipairs()` will work.
3. Handling of the actual drawing commands is delegated to a separate function `handle_content`.

The `handle_content` function is in fact an 'embedded' lpeg itself. It turns out that for this input, that is the simplest way of handling it. The overall structure of `handle_content` looks like this:

```
  function handle_content (resulttable, icontent)
    local function do_T  (...) parse_T (resulttable, ...) end
    local function do_T3 (...) parse_T3(resulttable, ...) end
    -- [omissions here]

    local xtc_T3 = P("T ")  * (Sp(oneI) * Sp(threeF) * Sp(oneI)
                  * Sp(twoF) * oneF/do_T3) * nl
    local xtc_T  = P("T ")  * (Sp(oneI) * Sp(twoF) * oneF/do_T)
                  * nl
    -- [more omissions here]
    local xtc_content = (xtc_SB + xtc_SE + xtc_T4 + xtc_T3 + xtc_T
                        + xtc_E4 + xtc_E + xtc_F4 + xtc_F3
                        + xtc_Q3 + xtc_G3 + xtc_L3 + xtc_W3
                        + xtc_D + xtc_S + xtc_C + xtc_P + xtc_B3
                        + xtc_Z )^1
    xtc_content:match(icontent)
  end
```

The new lpeg thing worth mentioning is that pattern matching is always greedy. In this case, that means that `xtc_T3` needs to appear in the `xtc_content` line before the `xtc_T`: match specifications that require a longer prefix need to always come before less precise specifications.

We have covered almost all the interesting bits of the `.xtc` parser already. The rest is just setting up the lpeg parser variable, and loading and parsing an actual file:

```
local xtc_statement = xtc_version + xtc_title1 + xtc_title2
                      + xtc_mapscale + xtc_roomsize + xtc_joint
                      + xtc_block + xtc_switchmotor + xtc_signal
                      + xtc_control + xtc_sensor + xtc_scale
                      + xtc_layers + xtc_layers_c + xtc_draw
                      + xtc_curve + xtc_cornu + xtc_turntable
                      + xtc_turnout + xtc_straight + xtc_car

local xtc_file = (xtc_statement + xtc_comment + emptyline
                  + endfile + xtc_rubbish )^1

xtc.parse = function (v)
    local data = io.loaddata(v)
    if data then
        layout = xtc.new_layout()
        xtc_file:match(data)
        if xtc.error then
            return nil, xtc.error
        end
        return layout
    end
    return nil, 'file not loadable'
end
```

The last bit worth mentioning is the definition of `xtc_rubbish` and its matching function:

```
local xtc_rubbish    = (P(1)^1 * rest/xtc.rubbish)

xtc.rubbish = function (a)
    if a and #a>0 then
        local v = string.split(a, '\n')
        xtc.error = xtc.error or ''
        xtc.error = xtc.error .. 'found: [' .. v[1] .. ']\n'
    end
end
```

This matches anything that was not explicitly mentioned previously and raises an error. While building up the parser and also to prepare for future file format changes, a function like that is a vital tool.

The MetaPost output

This will be the topic of one of next year's talks.