TEX Programming: The past, the present, and the future

Abstract
This article summarizes a recent thread on the ConTEXt mailing list.
(http://archive.contextgarden.net/thread/20090304.193503.1c42e4d5.en.html/)
To make the article interesting, I have changed the question and correspondingly modified the solutions.

Keywords
ConTEXt, luaTEX, TEX Programming

Suppose you want to typeset (in ConTEXt) all possible sums of roll of two dies, like this:

(+)	1	2	3	4	5	6
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	9	10
5	6	7	8	9	10	11
6	7	8	9	10	11	12

The mundane way to do this, *especially if you do not have too much time at hand,* is to type the whole thing by hand:

```
\bTABLE
  \bTR \bTD $(+)$ \eTD \bTD 1 \eTD \bTD 2 \eTD
       \bTD 3 \eTD \bTD 4  \eTD \bTD 5  \eTD \bTD 6  \eTD \eTR
  \bTR \bTD 1     \eTD \bTD 2 \eTD \bTD 3 \eTD
       \bTD 4 \eTD \bTD 5  \eTD \bTD 6  \eTD \bTD 7  \eTD \eTR
  \bTR \bTD 2     \eTD \bTD 3 \eTD \bTD 4 \eTD
       \bTD 5 \eTD \bTD 6  \eTD \bTD 7  \eTD \bTD 8  \eTD \eTR
  \bTR \bTD 3     \eTD \bTD 4 \eTD \bTD 5 \eTD
       \bTD 6 \eTD \bTD 7  \eTD \bTD 8  \eTD \bTD 9  \eTD \eTR
  \bTR \bTD 4     \eTD \bTD 5 \eTD \bTD 6 \eTD
       \bTD 7 \eTD \bTD 8  \eTD \bTD 9  \eTD \bTD 10 \eTD \eTR
  \bTR \bTD 5     \eTD \bTD 6 \eTD \bTD 7 \eTD
       \bTD 8 \eTD \bTD 9  \eTD \bTD 10 \eTD \bTD 11 \eTD \eTR
  \bTR \bTD 6     \eTD \bTD 7 \eTD \bTD 8 \eTD
       \bTD 9 \eTD \bTD 10 \eTD \bTD 11 \eTD \bTD 12 \eTD \eTR
\eTABLE
```

I am using Natural Tables since it is easy to configure its output (see `http://www.pragma-ade.com/general/manuals/enattab.pdf/`). For example, to get the effect shown above, I use the following setup:

```
\setupTABLE[each][each][width=2em,height=2em,align={middle,middle}]
\setupTABLE[r][1][background=color,backgroundcolor=gray]
\setupTABLE[c][1][background=color,backgroundcolor=gray]
```

Natural tables, however, are not the focus of this article. It is rather, what would you do if you are adventurous and have time at hand. The above is a repetitive task, so it should be possible to automate it. That will save typing errors (unless you make a mistake in your algorithm) and make the code reusable. In any ordinary programming language you could easily write something like the following pseudo code

```
  start_table
  start_table_row
     table_element("(+)")
      for y in [1..6] do
        table_elemnt(y)
  stop_table_row
  for x in [1..6] do
    start_table_row
      table_element(x)
      for y in [1..6] do
        table_element(x+y)
      end
    stop_table_row
  end
stop_table
```

But TeX is no ordinary programming language! Lets try to do this using ConTeXt's equivalent of a for-loop—`\dorecurse`

```
\bTABLE
 \bTR
  \bTD $(+)$ \eTD
  \dorecurse{6}
   {\bTD \recurselevel \eTD}
  \eTR
\dorecurse{6}
 {\bTR
     \bTD \recurselevel \eTD
     \edef\firstrecurselevel{\recurselevel}
  \dorecurse{6}
    {\bTD \the\numexpr\firstrecurselevel+\recurselevel \eTD}
  \eTR}
\eTABLE
```

This, however, does not work as expected because `\dorecurse` is not fully expandable. One way to get around this problem is to *expand* the appropriate parts of the body of `\dorecurse`

```
\bTABLE
 \bTR
  \bTD $(+)$ \eTD
  \dorecurse{6}
   {\expandafter \bTD \recurselevel \eTD}
  \eTR
\dorecurse{6}
 {\bTR
     \edef\firstrecurselevel{\recurselevel}
     \expandafter\bTD \recurselevel \eTD
  \dorecurse{6}
    {\expandafter\bTD
       \the\numexpr\firstrecurselevel+\recurselevel\relax
     \eTD}
  \eTR}
\eTABLE
```

Behold, the `\expandafter`! So, what is this expansion stuff, and why do we need `\expandafter`. TEX has a esoteric executing model, which was succinctly explained by David Kastrup in his TEX interview (`http://www.tug.org/interviews/interview-files/david-kastrup.html/`)

> "Instead, macros are used as a substitute for programming. TEX's macro expansion language is the only way to implement conditionals and loops, but the corresponding control variables can't be influenced by macro expansion (TEX's "mouth" in Knuth's terminology). Instead assignments must be executed by the back end (TEX's "stomach"). Stomach and mouth execute at different times and independently from one another. But it is not possible to solve nontrivial programming tasks with either: only the unholy chimera made from both can solve serious problems. ε-TEX gives the mouth a few more teeth and changes some of that, but the changes are not really fundamental: expansion still makes no assignments."

So, where do we add the `\expandafters`? It's simple, once you get the hang of it (Taco Hoekwater in a ConTEXt mailing list thread (`http://archive.contextgarden.net/message/20060702.141636.a57e6b68.en.html/`))

> "The trick to `\expandafter` is that you (normally) write it backwards until reaching a moment in time where TeX is not scanning an argument.
>
> Say you have a macro that contains some stuff in it to be typeset by `\type`:
>
> ```
> \def\mystuff{Some literal stuff}
> ```
>
> Then you begin with
>
> ```
> \type{\mystuff}
> ```
>
> but that obviously doesn't work, you want the final input to look like
>
> ```
> \type{Some literal stuff}
> ```
>
> Since `\expandafter` expands the token that follows after next token—whatever the next token is—you have to insert it backwards across the opening brace of the argument, like so:
>
> ```
> \type\expandafter{\mystuff}
> ```
>
> But this wouldn't work, yet: you are still in the middle of an expression (the `\type` expects an argument, and it gets `\expandafter` as it stands).
>
> Luckily, `\expandafter` itself is an expandable command, so you jump back once more and insert another one:
>
> ```
> \expandafter\type\expandafter{\mystuff}
> ```
>
> Now you are on 'neutral ground', and can stop backtracking. *Easy, once you get the hang of it.*"

If you do not get the hang of it, relax. ConTEXt provides a command \expanded that expands its arguments.

```
\bTABLE
 \bTR
  \bTD $(+)$ \eTD
  \dorecurse{6}
    {\expanded{\bTD \recurselevel \eTD}}
  \eTR
  \dorecurse{6}
    {\bTR
       \expanded{\bTD \recurselevel \eTD}
       \edef\firstrecurselevel{\recurselevel}
     \dorecurse{6}
       {\expanded{\bTD
        \the\numexpr\firstrecurselevel+\recurselevel\relax \eTD}}
    \eTR}
\eTABLE
```

Using \expanded is easier than using \expandafter, but you still need to understand TEX's expansion mechanism to get it right. For example, if you try

```
  ...
  \dorecurse{6}
    {\expaned{\bTR
      \bTD \recurselevel \eTD
       \edef\firstrecurselevel{\recurselevel}
     \dorecurse{6}
       {\expanded{\bTD
        \the\numexpr\firstrecurselevel+\recurselevel\relax \eTD}}
    \eTR}}
  ...
```

you will get all sorts of TEX errors, and you need to sprinkle \noexpand at correct places to get it to work. So, \expanded is not a silver bullet.

In the above mention mailing list thread, Wolfgang Schuster posted a much neater solution.

```
\bTABLE
 \bTR
  \bTD $(+)$ \eTD
  \dorecurse{6}
   {\bTD #1 \eTD}
  \eTR
\dorecurse{6}
 {\bTR
     \bTD #1 \eTD
  \dorecurse{6}
    {\bTD \the\numexpr#1+##1 \eTD}
  \eTR}
\eTABLE
```

This makes TEX disguise as a **normal** programming language. But only TEX wizards like Wolfgang can discover such solutions. You need to know the TeX digestive system inside out to even attempt something like this. Inspired by Wolfgang's solution, I tried the same thing with ConTEXt's lesser known for loops

```
\bTABLE
  \bTR
    \bTD $(+)$ \eTD
    \for \y=1 \to 6 \step 1 \do
      {\bTD #1 \eTD}
  \eTR
  \for \x=1 \to 6 \step 1 \do
  {\bTR
     \bTD #1 \eTD
    \for \y=1 \to 6 \step 1 \do
    {\bTD \the\numexpr#1+##1 \eTD}
  \eTR}
\eTABLE
```

Is your head hurting. Don't worry. luaTEX provides hope that normal users can do simple programming tasks. Luigi Scarso posted the following code:

```
\startluacode
    tprint = function(s) tex.sprint(tex.ctxcatcodes,s) end
    tprint('\\bTABLE')
    tprint('\\bTR')
    tprint('\\bTD $(+)$ \\eTD')
    for y = 1,6 do
      tprint('\\bTD ' .. y .. '\\eTD')
    end
    tprint('\\eTR')
    for x = 1,6 do
      tprint('\\bTR')
      tprint('\\bTD ' .. x .. '\\eTD')
      for y = 1,6 do
        tprint('\\bTD' .. x+y .. '\\eTD')
      end
      tprint('\\eTR')
    end
    tprint('\\eTABLE')
\stopluacode
```

Finally luaTEX offers a simple way of implementing simple algorithms inside TEX. There is no need to know TEX's digestive system. Write code as you would write in any other programing language!

If you are a TEX programming guru who can keep track of TEX's expansion mechanism, don't fear luaTEX. There are other options for you: mix TEX and MetaPost.

```
\let\normalbTABLE\bTABLE
\let\normaleTABLE\eTABLE

\unexpanded\def\bTABLE{\normalbTABLE}
\unexpanded\def\eTABLE{\normaleTABLE}

\unexpanded\def\dobTR{\dodoubleempty\parseTR}
\unexpanded\def\dobTD{\dodoubleempty\parseTD}
\unexpanded\def\dobTH{\dodoubleempty\parseTH}
\unexpanded\def\dobTN{\dodoubleempty\parseTN}

\let\bTR\dobTR
\let\bTD\dobTD
\let\bTH\dobTH
```

```
\let\bTN\dobTN

\startMPcode
  string table ;
  table = "\bTABLE \bTR \bTD $(+)$ \eTD" &
  for y = 1 upto 6 :
    "\bTD " & decimal y & "\eTD " &
  endfor
  "\eTR " &
  for x = 1 upto 6 :
    "\bTR \bTD " & decimal x & "\eTD " &
    for y = 1 upto 6 :
      "\bTD " & decimal (x+y) & "\eTD " &
    endfor
    "\eTR" &
  endfor
  "\eTABLE" ;
  label(textext(table), origin) ;
\stopMPcode
```

Aditya Mahajan
adityam@umich.edu