
SciTE
IN CONTEXT MkIV

About SCITE

This manual is under (re)construction.

For a long time at Pragma ADE we used TEXedit, an editor we'd written in Modula. It had some project management features and recognized the project structure in ConTEXt documents. Later we rewrote this to a platform independent reimplementation called TEXwork written in Perl/Tk (not to be confused with the editor with the plural name).

In the beginning of the century I can into SciTE, written by Neil Hodgson. Although the mentioned editors provide some functionality not present in SciTE we decided to use that editor because it frees us from maintaining our own. I ported our TEX and MetaPost (line based) syntax highlighting to SciTE and got a lot of others for free.

After a while I found out that there was an extension interface written in Lua. I played with it and wrote a few extensions too. This pleasant experience later triggered the LuaTEX project.

A decade into the century SciTE got another new feature: you can write dynamic external lexers in Lua using lpeg. As in the meantime ConTEXt has evolved in a TEX/Lua hybrid, it made sense to look into this. The result is a couple of lexers that suit TEX, MetaPost and Lua usage in ConTEXt MkIV. As we also use xml as input and output format a lexer for xml is also provided. And because pdf is one of the backend formats lexing of pdf is also implemented.[1]

In the ConTEXt (standalone) distribution you will find the relevant files under:

```
<texroot>/tex/texmf-context/context/data/scite
```

Normally a user will not have to dive into the implementation details but in principle you can tweak the properties files to suit your purpose.

[1] In the process some of the general lexing framework was adapted to suit our demands for speed. We ship these files as well.

The look and feel

The color scheme that we use is consistent over the lexers but we use more colors that in the traditional lexing. For instance, TEX primitives, low level TEX commands, TEX constants, basic file structure related commands, and user commands all get a different treatment. When spell checking is turned on, we indicate unknown words, but also words that are known but might need checking, for instance because they have an uppercase character. In figure 1 we some of that in practice.

Installing SCITE

Installing SciTE is straightforward. We are most familiar with MS Windows but for other operating systems installation is not much different. First you need to fetch the archive from:

```
www.scintilla.org
```

The MS Windows binaries are zipped in `wscite.zip`, and you can unzip this in any directory you want as long as you make sure that the binary ends up in your path or as shortcut on your desktop. So, say that you install SciTE in:

```
c:\data\system\scite\wscite
```

You need to add this path to your local path definition. Installing SciTE to some known place has the advantage that you can move it around. There are no special dependencies on the operating system.

Next you need to install the lpeg lexers.[2] These can be fetched from:

```
code.google.com/p/scintilla
```

On windows you need to copy the `lexers` subfolder to the `wscite` folder. For Linux the place depends on the distribution and I just copy them in the same path as where the regular properties files live.

For Unix, one can take a precompiled version as well. Here we need to split the set of files into:

[2] Versions later than 2.11 will not run on Windows 2K. In that case you need to comment the external lexer import.

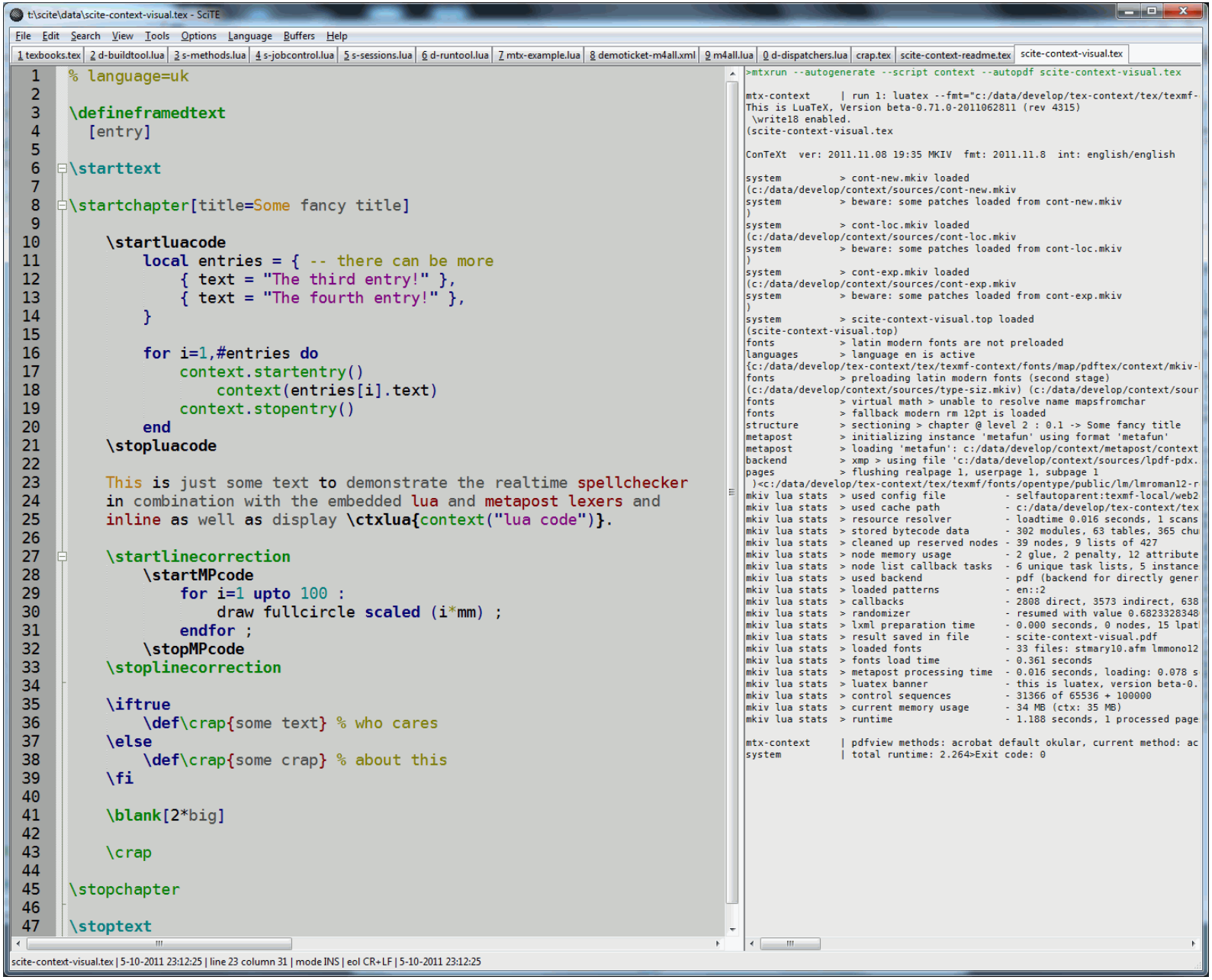

Figure 1 Nested lexers in action.

```
/usr/bin
/usr/share/scite
```

The second path is hard coded in the binary.

If you want to use ConTEXt, you need to copy the relevant files from

```
<texroot>/tex/texmf-context/context/data/scite
```

to the path were SciTE keeps its property files (`*.properties). There is a file called \type{SciteGlobal.properties}. At the end of that file` on windows it is in the path where the Scite binary) you then add a line to the end:

```
import scite-context-user
```

You need to restart SciTE in order to see if things work out as expected.

Disabling the external lexer in a recent SciTE is somewhat tricky. In that case the end of that file looks like:

```
imports.exclude=scite-context-external
import *
import scite-context-user
```

In any case you need to make sure that the user file is loaded last.

After this, things should run as expected (given that TEX runs at the console as well).

Fonts

The configuration file defaults to the Dejavu fonts. These free fonts are part of the ConTEXt suite (also known as the standalone distribution). Of course you can fetch them from `http://dejavu-fonts.org` as well. You have to copy them to where your operating system expects them. In the suite they are available in

```
<contextroot>/tex/texmf/fonts/truetype/public/dejavu
```

An alternative approach

If for some reason you prefer not to mess with property files in the main SciTE path, you can follow a different route and selectively copy files to places.

The following files are needed for the lpeg based lexer:

```
lexers/scite-context-lexer.lua
lexers/scite-context-lexer-tex.lua
lexers/scite-context-lexer-mps.lua
lexers/scite-context-lexer-lua.lua
lexers/scite-context-lexer-cld.lua
lexers/scite-context-lexer-txt.lua
lexers/scite-context-lexer-xml*.lua
lexers/scite-context-lexer-pdf*.lua

lexers/context/data/scite-context-data-tex.lua
lexers/context/data/scite-context-data-context.lua
lexers/context/data/scite-context-data-interfaces.lua
lexers/context/data/scite-context-data-metapost.lua
lexers/context/data/scite-context-data-metafun.lua

lexers/themes/scite-context-theme.lua
```

The data files are needed because we cannot access property files from within the lexer. If we could open a file we could use the property files instead.

These files go to the `lexers` subpath in your SciTE installation. Normally this sits in the binary path. The following files provide some extensions. On windows you can copy these files to the path where the SciTE binary lives.

```
scite-ctx.lua
```

Because property files can only be loaded from the same path where the (user) file loads them you need to copy the following files to the same path where the loading is defined:

```
scite-context.properties
```

```
scite-context-internal.properties
scite-context-external.properties

scite-pragma.properties

scite-tex.properties
scite-metapost.properties

scite-context-data-tex.properties
scite-context-data-context.properties
scite-context-data-interfaces.properties
scite-context-data-metapost.properties
scite-context-data-metafun.properties

scite-ctx.properties
scite-ctx-context.properties
scite-ctx-example.properties
```

On Windows these go to:

```
c:/Users/YourName
```

Next you need to add this to:

```
import scite-context
import scite-context-internal
import scite-context-external
import scite-pragma
```

to the file:

```
SciTEUser.properties
```

Of course the pragma import is optional. You can comment either the internal or external variant but there is no reason not to keep them both.

Extensions

Just a quick not to some extensions. If you select a part of the text (normally you do this with the shift key pressed) and you hit `Shift-F11`, you get a menu with some options. More (robust) ones will be provided at some point.

Spell checking

If you want to have spell checking, you need have files with correct words on each line. The first line of a file determines the language:

```
% language=uk
```

When you use the external lexers, you need to provide some files. Given that you have a text file with valid words only, you can run the following script:

```
mtxrun --script scite --words nl uk
```

This will convert files with names like `spell-nl.txt` into Lua files that you need to copy to the `lexers/data` path. Spell checking happens realtime when you have the language directive (just add a bogus character to disable it). Wrong words are colored red, and words that might have a case problem are colored orange. Recognized words are greyed and words with less than three characters are ignored.

In the case of internal lexers, the following file is needed:

```
spell-uk.txt
```

If you use the traditional lexer, this file is taken from the path determined by the environment variable:

```
CTXSPELLPATH
```

As already mentioned, the lpeg lexer expects them in the data path. This is because the Lua instance that does the lexing is rather minimalistic and lacks some libraries as well as cannot access the main SciTE state.

Spell checking in `txt` files is enabled by adding a first line:

```
[#!-%] language=uk
```

The first character on that line is one of the four mentioned between square brackets. So,

```
# language=uk
```

should work. For xml files there are two methods. You can use the following (at the start of the file):

```
<?xml ... language="uk" ?>
```

But probably better is to use the next directive just below the usual xml marker line:

```
<?context-xml-directive editor language uk ?>
```

Interface selection

In a similar fashion you can drive the interface checking:

```
% interface=nl
```

Property files

The internal lexers are controlled by the property files while the external ones are steered with themes. Unfortunately there is hardly any access to properties from the external lexer code nor can we consult the file system and/or run programs like `mtxrun`. This means that we cannot use configuration files in the ConTEXt distribution directly. Hopefully this changes with future releases.

The external lexers

These are the more advanced. They provide more detail and the ConTEXt lexer also supports nested MetaPost and Lua. Currently there is no detailed configuration but this might change once they are stable.

The external lexers operate on documents while the internal ones operate on lines. This can make the external lexers slow on large documents. We've optimized the code somewhat for speed and memory consumption but there's only so much one can do. While lexing each change in style needs a small table but allocating and garbage collecting many small tables comes at a price. Of course in practice this probably gets unnoticed.[3]

[3] I wrote the code in 2011 on a more than 5 years old Dell M90 laptop, so I suppose that speed is less an issue now.

In principle the external lexers can be used with `textadept` which also uses `scintilla`. Actually, support for lpeg lexing originates in `textadept`. Currently `textadept` lacks a couple of features I like about SciTE (for instance it has no realtime logpane) and it's also still changing. At some point the ConTEXt distribution might ship with files for `textadept` as well.

The external lpeg lexers work okay with the MS Windows and linux versions of SciTE, but unfortunately at the time of writing this, the Lua library that is needed is not available for the MacOSX version of SciTE. Also, due to the fact that the lexing framework is rather isolated, there are some issues that cannot be addressed in the properly, at least not currently.

In addition to ConTEXt and MetaFun lexing a Lua lexer is also provided so that we can handle ConTEXt Lua Document (cld) files too. There is also an xml lexer. This one also provides spell checking. The pdf lexer tries to do a good job on pdf files, but it has some limitations. There is also a simple text file lexer that does spell checking.

Don't worry if you see an orange rectangle in your TEX or xml document. This indicates that there is a special space character there, for instance `0xA0`, the nonbreakable space. Of course we assume that you use utf8 as input encoding.

The internal lexers

SciTE has quite some built in lexers. A lexer is responsible for highlighting the syntax of your document. The way a TEX file is treated is configured in the file:

```
tex.properties
```

You can edit this file to your needs using the menu entry under `options` in the top bar. In this file, the following settings apply to the TEX lexer:

```
lexer.tex.interface.default=0
lexer.tex.use.keywords=1
lexer.tex.comment.process=0
lexer.tex.auto.if=1
```

The option `lexer.tex.interface.default` determines the way keywords are highlighted. You can control the interface from your document as well, which makes more sense that editing the configuration file each time.

```
% interface=all|tex|nl|en|de|cz|it|ro|latex
```

The values in the properties file and the keywords in the preamble line have the following meaning:

0	all	all commands (preceded by a backslash)
1	tex	TEX, ε-TEX, pdfTEX, Omega primitives (and macros)
2	nl	the dutch ConTEXt interface
3	en	the english ConTEXt interface
4	de	the german ConTEXt interface
5	cz	the czech ConTEXt interface
6	it	the italian ConTEXt interface
7	ro	the romanian ConTEXt interface
8	latex	LATEX (apart from packages)

The configuration file is set up in such a way that you can easily add more keywords to the lists. The keywords for the second and higher interfaces are defined in their own properties files. If you're curious about the way this is configures, you can peek into the property files that start with `scite-context`. When you have ConTEXt installed you can generate configuration files with

```
mtxrun --script interface --scite
```

You need to make sure that you move the result to the right place so best not mess around with this command and use the files from the distribution.

Back to the properties in `tex.properties`. You can disable keyword coloring alltogether with:

```
lexer.tex.use.keywords=0
```

but this is only handy for testing purposes. More interesting is that you can influence the way comment is treated:

```
lexer.tex.comment.process=0
```

When set to zero, comment is not interpreted as TEX code and it will come out in a uniform color. But, when set to one, you will get as much colors as a TEX source. It's a matter of taste what you choose.

The lexer tries to cope with the TEX syntax as good as possible and takes for instance care of the funny ^^ notation. A special treatment is applied to so called \if's:

```
lexer.tex.auto.if=1
```

This is the default setting. When set to one, all \ifwhatever's will be seen as a command. When set to zero, only the primitive \if's will be treated. In order not to confuse you, when this property is set to one, the lexer will not color an \ifwhatever that follows an \newif.

The MetaPost lexer

The MetaPost lexer is set up slightly different from its TEX counterpart, first of all because MetaPost is more a language that TEX. As with the TEX lexer, we can control the interpretation of identifiers. The MetaPost specific configuration file is:

```
metapost.properties
```

Here you can find properties like:

```
lexer.metapost.interface.default=1
```

Instead of editing the configuration file you can control the lexer with the first line in your document:

```
% interface=none|metapost|mp|metafun
```

The numbers and keywords have the following meaning:

0	none	no highlighting of identifiers
1	metapost or mp	MetaPost primitives and macros
2	metafun	MetaFun macros

Similar to the TEX lexer, you can influence the way comments are handled:

```
lexer.metapost.comment.process=1
```

This will interpret comment as MetaPost code, which is not that useful (opposite to TEX, where documentation is often coded in TEX).

The lexer will color the MetaPost keywords, and, when enabled also additional keywords (like those of MetaFun). The additional keywords are colored and shown in a slanted font.

The MetaFun keywords are defined in a separate file:

```
metafun-scite.properties
```

You can either copy this file to the path where you global properties files lives, or put a copy in the path of your user properties file. In that case you need to add an entry to the file `SciTEUser.properties`:

```
import metafun-scite
```

The lexer is able to recognize `btex–etex` and will treat anything in between as just text. The same happens with strings (between `"`). Both act on a per line basis.

Using ConTEXt

When `mtxrun` is in your path, ConTEXt should run out of the box. You can find `mtxrun` in:

```
<contextroot>/tex/texmf-mswin/bin
```

or in a similar path that suits the operating system that you use.

When you hit `CTRL-12` your document will be processed. Take a look at the `Tools` menu to see what more is provided.

Extensions (using LUA)

When the Lua extensions are loaded, you will see a message in the log pane that looks like:

```
-  see scite-ctx.properties for configuring info

-  ctx.spellcheck.wordpath set to ENV(CTXSPELLPATH)
-  ctxspellpath set to c:\data\develop\context\spell
-  ctx.spellcheck.wordpath expands to c:\data\develop\context\spell
```

```
-  ctx.wraptext.length is set to 65
-  key bindings:

Shift + F11   pop up menu with ctx options

Ctrl  + B     check spelling
Ctrl  + M     wrap text (auto indent)
Ctrl  + R     reset spelling results
Ctrl  + I     insert template
Ctrl  + E     open log file

-  recognized first lines:

xml   <?xml version='1.0' language='nl'
tex   % language=nl
```

This message tells you what extras are available.

Templates

There is an experimental template mechanism. One option is to define templates in a properties file. The property file `scite-ctx-context` contains definitions like:

```
command.25.$(file.patterns.context)=insert_template \
$(ctx.template.list.context)

ctx.template.list.context=\
    itemize=structure.itemize.context|\
    tabulate=structure.tabulate.context|\
    natural TABLE=structure.TABLE.context|\
    use MP graphic=graphics.usemp.context|\
    reuse MP graphic=graphics.reusemp.context|\
    typeface definition=fonts.typeface.context

ctx.template.structure.itemize.context=\
\startitemize\n\
\item ?\n\
\item ?\n\
```

```
\item ?\n\
\stopitemize\n
```

The file `scite-ctx-example` defines xml variants:

```
command.25.$(file.patterns.example)=insert_template \
$(ctx.template.list.example)

ctx.template.list.example=\
    bold=font.bold.example|\
    emphasized=font.emphasized.example|\
    |\
    inline math=math.inline.example|\
    display math=math.display.example|\
    |\
    itemize=structure.itemize.example

ctx.template.structure.itemize.example=\
<itemize>\n\
<item>?</item>\n\
<item>?</item>\n\
<item>?</item>\n\
</itemize>\n
```

For larger projects it makes sense to keep templates with the project. In one of our projects we have a directory in the path where the project files are kept which holds template files:

```
..../ctx-templates/achtergronden.xml
..../ctx-templates/bewijs.xml
```

One could define a template menu like we did previously:

```
ctx.templatelist.example=\
    achtergronden=mathadore.achtergronden|\
    bewijs=mathadore.bewijs|\

ctx.template.mathadore.achtergronden.file=smt-achtergronden.xml
ctx.template.mathadore.bewijs.file=smt-bewijs.xml
```

However, when no such menu is defined, we will automatically scan the directory and build the menu without user intervention.

Using SCITE

The following keybindings are available in SciTE. Most of this list is taken from the on-line help pages.

keybinding	**meaning (taken from the** SciTE **help file)**
`Ctrl+Keypad+`	magnify text size
`Ctrl+Keypad-`	reduce text size
`Ctrl+Keypad/`	restore text size to normal
`Ctrl+Keypad*`	expand or contract a fold point
`Ctrl+Tab`	cycle through recent files
`Tab`	indent block
`Shift+Tab`	dedent block
`Ctrl+BackSpace`	delete to start of word
`Ctrl+Delete`	delete to end of word
`Ctrl+Shift+BackSpace`	delete to start of line
`Ctrl+Shift+Delete`	delete to end of line
`Ctrl+Home`	go to start of document; `Shift` extends selection
`Ctrl+End`	go to end of document; `Shift` extends selection
`Alt+Home`	go to start of display line; `Shift` extends selection
`Alt+End`	go to end of display line; `Shift` extends selection
`Ctrl+F2`	create or delete a bookmark
`F2`	go to next bookmark
`Ctrl+F3`	find selection
`Ctrl+Shift+F3`	find selection backwards
`Ctrl+Up`	scroll up
`Ctrl+Down`	scroll down
`Ctrl+C`	copy selection to buffer
`Ctrl+V`	insert content of buffer
`Ctrl+X`	copy selection to buffer and delete selection
`Ctrl+L`	line cut
`Ctrl+Shift+T`	line copy

Ctrl+Shift+L	line delete
Ctrl+T	line transpose with previous
Ctrl+D	line duplicate
Ctrl+K	find matching preprocessor conditional, skipping nested ones
Ctrl+Shift+K	select to matching preprocessor conditional
Ctrl+J	find matching preprocessor conditional backwards, skipping nested ones
Ctrl+Shift+J	select to matching preprocessor conditional backwards
Ctrl+[	previous paragraph; Shift extends selection
Ctrl+]	next paragraph; Shift extends selection
Ctrl+Left	previous word; Shift extends selection
Ctrl+Right	next word; Shift extends selection
Ctrl+/	previous word part; Shift extends selection
Ctrl+\	next word part; Shift extends selection

Affiliation

author	Hans Hagen
copyright	PRAGMA ADE, Hasselt NL
more info	www.pragma-ade.com
	www.contextgarden.net
version	November 11, 2011