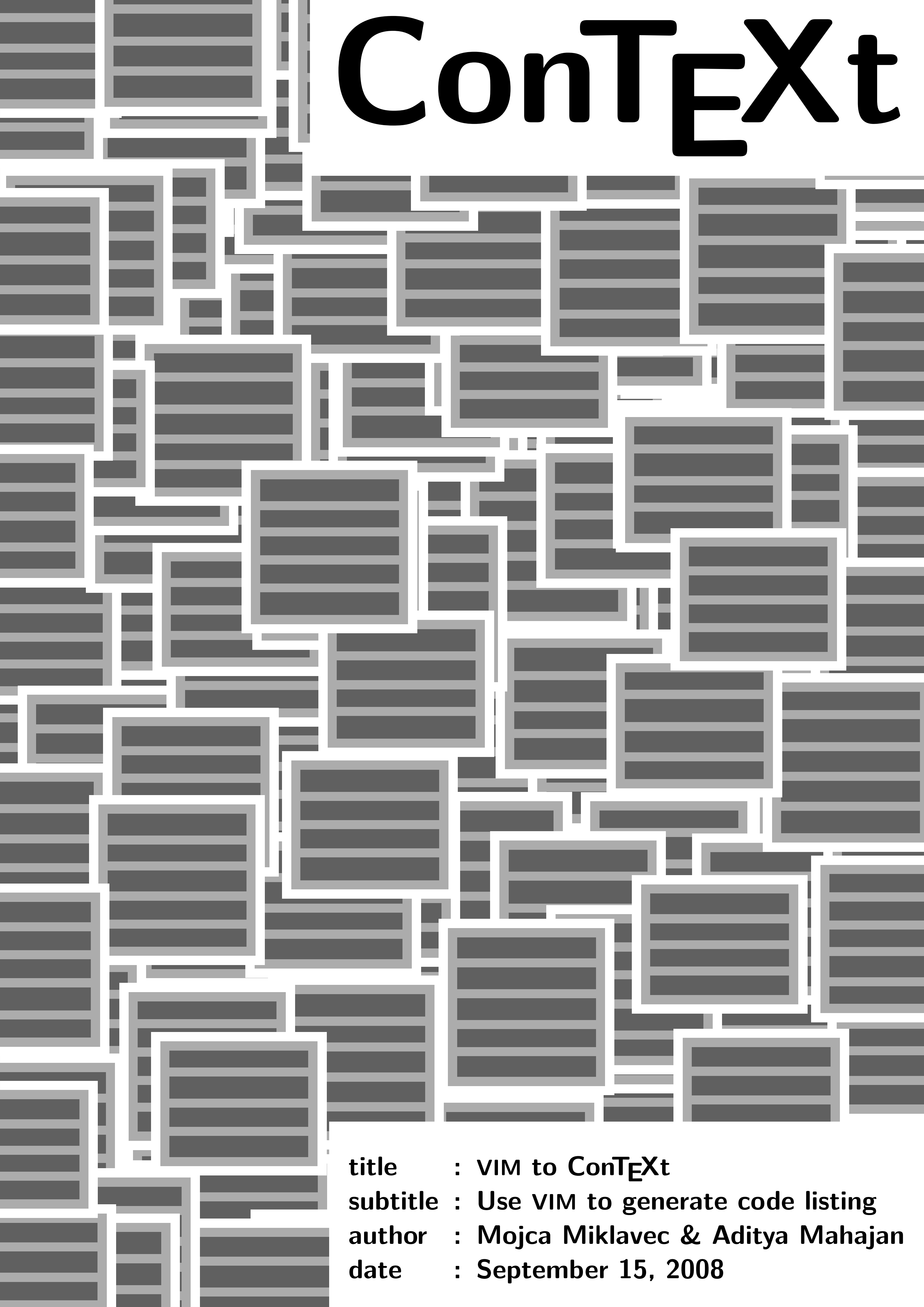
ConTEXt



title	:	VIM to ConTEXt
subtitle	:	Use VIM to generate code listing
author	:	Mojca Miklavec & Aditya Mahajan
date	:	September 15, 2008

1 User Manual

CONTEXT has an excellent pretty printing capabilities for many languages. The code for pretty printing is written in TEX, and due to catcode jugglery verbatim typesetting is perhaps the trickiest part of TEX. This makes it difficult for a “normal” user to define syntax highlighting rules for a new language. This module, takes the onus of defining syntax highlighting rules away from the user and uses VIM editor to generate the syntax highlighting. There is a helper `2context.vim` script to do the syntax parsing in VIM. This is a stop-gap method, and hopefully with LUATEX, things will be much easier.

The main macro of this module is `\definevimtyping`. The best way to explain it is by using an example. Suppose you want to pretty print ruby code in CONTEXT. So you can do

```
\definevimtyping [RUBY]  [syntax=ruby]
```

after which you can get ruby highlighting by

```
\startRUBY
....
\stopRUBY
```

For example

```
#!  /usr/bin/ruby
# This is my first ruby program
puts "Hello World"
```

This was typed as

```
\definevimtyping [RUBY] [syntax=ruby]

\startRUBY
#! /usr/bin/ruby
# This is my first ruby program
puts "Hello World"
\stopRUBY
```

The typing can be setup using `\setupvimtyping`.

```
\setupvimtyping [..,.=.,..]
                     *

*   syntax       =  IDENTIFIER
    colorscheme  =  IDENTIFIER
    space        =  yes  on  no
    tab          =  NUMBER
    start        =  NUMBER
    stop         =  NUMBER
    numbering    =  yes  no
    step         =  NUMBER
    numberstyle  =
    numbercolor  =  IDENTIFIER
    before       =  COMMAND
    after        =  COMMAND
```

Here `syntax` is the syntax file in VIM for the language highlighting that you want. See `:he syntax.txt` inside VIM for details. `colorscheme` provides the syntax highlighting for various regions. Right now, two colorschemes are defined. The `default` colorscheme is based on on `ps_color.vim` colorscheme in VIM, and the `blackandwhite` colorscheme is based on `print_bw.vim`. If there is a particular colorscheme that you will like, you can convert it into CONTEXT. `space=(yes|on|no)` makes the space significant, visible, and insignificant respectively. `tab` specifies the number of spaces a tab is equivalent to. It's default value is 8. `start` and `stop` specify which lines to read from a file. These options only make sense for highlighting files and should not to be set by `\setupvimtyping`. `numbering` enables line numbering, and `step` specifies which lines are numbered. `numberstyle` and `numbercolor` specify the style and color of line numbers. By default the numbers are placed on the left. The location of the numbers can be configured using `numbercommand` option.

A new typing region can be define using `\definevimtyping`.

```
\definevimtyping [.1.] [.2.]
                        OPTIONAL

1   IDENTIFIER
2   inherits from \setupvimtyping
```

Minor changes in syntax highlighting can be made easily. For example, Mojca likes 'void' to be bold in C programs. This can be done as follows

```
\definevimtyping [C] [syntax=c,numbering=on]

\startvimcolorscheme[default]

\definevimsyntax
  [Type]
  [style=boldmono]

\definevimsyntax
  [PreProc]
  [style=slantedmono]

\stopvimcolorscheme

\startC
#include <stdio.h>
#include <stdlib.h>

void main()
{
   printf("Hello World\n") ;
   return;
}
\stopC
```

which gives

```
#include <stdio.h>
#include <stdlib.h>

void main()
{
   printf("Hello World\n") ;
   return;
}
```

The second command provided by this module is `\definetypevimfile` for typesetting files. The syntax of this command is

```
\definetypevimfile [.1.] [.2.]
                         OPTIONAL

1   IDENTIFIER
2   inherits from \setupvimtyping
```

For example, to pretty print a ruby file you can do

```
\definetypevimfile[typeRUBY] [syntax=ruby]
```

after which one can use

```
\typeRUBY[option]{rubyfile}
```

We hope that this is sufficient to get you started. The rest of this document gives the implementation details of the module. If you want to change something, read ahead.

2 Module Details

The syntax highlighting of the source here is done using `t-vim` module. There is a bug in the module due to which line numberings for different filetypes use the same counter. In the source round–about method to correct this. Right now, in case someone needs this module for numbering more than one filetype, let me know, and I will try to iron out the bug.

```
\writestatus  {loading}   {Context Module for ViM Sytax Highlighting}

\startmodule[vim]

\unprotect

\definesystemvariable {vs}  % Vim Syntax
```

First of all we take care of bold monotype. By default, CONTEXT uses latin modern fonts. If you want to get bold monotype in latin modern, you need to use `modern-base` typescript. For example:

```
\usetypescript[modern-base][texnansi] \setupbodyfont[modern]
\starttext
{\tt\bf This is bold monotype}
\stoptext
```

CONTEXT does not provide any style alternative for bold monotype and slanted monotype, so we provide one here. These will only work if your font setup knows about bold and slanted monotype.

```
\definealternativestyle [\v!bold\v!mono,\v!mono\v!bold]       [\ttbf] []
\definealternativestyle [\v!slanted\v!mono,\v!mono\v!slanted] [\ttsl] []
```

`\startvimc..` To start a new vim colorscheme.

```
\def\startvimcolorscheme[#1]%
  {\pushmacro\vimcolorscheme
   \edef\vimcolorscheme{#1}}

\def\stopvimcolorscheme
  {\popmacro\vimcolorscheme}
```

`\definevim..`
`\definevim..`
These macros should always occur inside a `\startvimcolorschme ...\stopvimcolorscheme` pair. The `\definevimsyntax` macro defines syntax highlighting rules for VIM's syntax highlighting regions. It takes three arguments `style`, `color` and `command`. The most common VIM syntax highlighting regions are defined in the end of this file. The `\definevimsyntaxsynonyms` macro just copies the settings from another syntax highlighting region.

```
\def\definevimsyntax
  {\dodoubleargumentwithset\dodefinevimsyntax}

\def\dodefinevimsyntax[#1]% [#2]
  {\getparameters[\??vs\vimcolorscheme#1]} %[#2]
```

```
\def\definevimsyntaxsynonyms
  {\dodoubleargumentwithset\dodefinevimsyntaxsynonyms}
```

```
\def\dodefinevimsyntaxsynonyms[#1][#2]%
  {\copyparameters[\??vs\vimcolorscheme#1][\??vs\vimcolorscheme#2]
                  [\c!style,\c!color,\c!command]}
```

`\vimsyntax` This is just a placeholder macro. The `2context.vim` script marks the highlightin reigions by `\s[...]{...}`. While typing the generated files, we locally redefine `\s` to `\vimsyntax`.

```
\def\vimsyntax[#1]#2%
  {\dostartattributes{\??vs\vimcolorscheme Normal}\c!style\c!color\empty%
   \dostartattributes{\??vs\vimcolorscheme #1}\c!style\c!color\empty%
   \getvalue{\??vs\vimcolorscheme #1\c!command}{#2}%
    \dostopattributes%
    \dostopattributes}
```

`\setupvimt..` `\typevimfile` There are three settings for `\setupvimtyping`: `syntax`, which tells VIM which syntax rules to use; `tab`, which sets the `tabstop` in VIM; and `space` which takes care of spaces.

`\typevimfile` macro basically calls VIM with appropriate settings and sources the `2context.vim` script. The result is slow, because parsing by VIM is slow. Do not use this method for anything larger than a few hundred lines. For large files, one option is to pre–prase them, and then typeset the result. We have not provided any interface for that, but it is relatively easy to implement.

Taking care of line–numbering is more tricky. We could not get `\setuplinenumbering` to work properly, so implement our own line–numbering mechanism. This is a bit awkward, since it places line–number after each `^M` in the source file. So, if the source code line is larger than one typeset line, the line number will be on the second line. To do it correctly, we need to read lines from the vimsyntax file one-by-one. Our own mechanism for line–numbering is plain. Unlike CONTEXT's core verbatim highlighting, multiple blank lines are displayed and numbered.

```
\def\setupvimtyping
  {\dosingleargument\getparameters[\??vs]}
```

```
\def\typevimfile
  {\dosingleempty\dotypevimfile}
```

```
\def\notypevimfile[#1][#2]#3%
  {\dotypevimfile[#1,#2]{#3}}
```

```
\def\dotypevimfile[#1]#2%
  {\doiffileelse{#2}
   {\dodotypevimfile[#1]{#2}}
   {\reporttypingerror{#2}}}
```

```
\def\saveandtypevimfile[#1]%
```

```
  {\savevimbuffer
   \dotypevimfile[#1]{\TEXbufferfile{vimsyntax}}}

\let\savevimbuffer\donothing

\beginLUATEX

\def\savevimbuffer{\savebuffer[vimsyntax]}

\endLUATEX

\def\dodotypevimfile[#1]#2%
  {\@@vsbefore
   \bgroup
   \initializevimtyping{#1}
   \runvimsyntax{#2}
    % The strut is needed for the output to be the same when not using
    % numbering.  Otherwise, multiple par's are ignored.  We need to figure out
    % a mechanism to imitate this behaviour even while using line numbering.
    \strut%else the first line is shifted to the left
    \input #2-vimsyntax.tmp\relax%
   \egroup
   \@@vsafter}

\makecounter{vimlinenumber}

\def\doplacevimlinenumber
  {%Always place the first linenumber
   \showvimlinenumber
   %Calculate step in futute
   \let\placevimlinenumber\dodoplacevimlinenumber
   \pluscounter{vimlinenumber}}

\def\dodoplacevimlinenumber
  {\ifnum\numexpr(\countervalue{vimlinenumber}/\@@vsstep)*\@@vsstep\relax=%
         \numexpr\countervalue{vimlinenumber}\relax
      \showvimlinenumber
  \fi
   \pluscounter{vimlinenumber}}

\def\showvimlinenumber
  {\@@vsnumbercommand
      {\dostartattributes\??vs\c!numberstyle\c!numbercolor\empty
       \countervalue{vimlinenumber}
       \dostopattributes}}
```

```
\def\initializevimtyping#1
  {\setupvimtyping[#1]
   %Make sure that stop is not empty
   \doifempty{\@@vsstop}{\setvalue{\@@vsstop}{0}}
   \doifelse{\@@vsstart}{\v!continue}
    {\setvalue{@@vsstart}{\countervalue{vimlinenumber}}}
    {\setcounter{vimlinenumber}{\doifnumberelse{\@@vsstart}{\@@vsstart}{1}}}
   \whitespace
  %\page[\v!preference]} gaat mis na koppen, nieuw:  later \nobreak
   \setupwhitespace[\v!none]%
   \obeylines
   \ignoreeofs
   \ignorespaces
   \activatespacehandler\@@vsspace
   \let\s=\vimsyntax
   \def\tab##1{\dorecurse{##1}{\space}}% TODO: allow customization
   \def\vimcolorscheme{\@@vscolorscheme}
   \processaction[\@@vsnumbering]
   [     \v!on=>\let\placevimlinenumber\doplacevimlinenumber,
        \v!off=>\let\placevimlinenumber\relax,
    \s!unknown=>\let\placevimlinenumber\relax,
    \s!default=>\let\placevimlinenumber\relax,
   ]
   \def\obeyedline{\placevimlinenumber\par\strut}
   }


\def\shellescapedquote{\letterbackslash\letterdoublequote}


\def\runvimsyntax#1
   {\executesystemcommand
      {mtxrun --verbose --noquote bin:vim
          "-u NONE  % No need to read unnessary configurations
           -e       % run in ex mode
           -C       % Set compatibile
           -n       % No swap
           -c \shellescapedquote set tabstop=\@@vstab \shellescapedquote\space
           -c \shellescapedquote syntax on\shellescapedquote\space
           -c \shellescapedquote set syntax=\@@vssyntax\shellescapedquote\space
           -c \shellescapedquote let contextstartline=\@@vsstart\shellescapedquote\space
           -c \shellescapedquote let contextstopline=\@@vsstop\shellescapedquote
\space
           -c \shellescapedquote source kpse:2context.vim\shellescapedquote\space
           -c \shellescapedquote wqa\shellescapedquote\space
            \shellescapedquote#1\shellescapedquote\space "}}
```

\definetyp.. This macro allows you to define new file typing commands. For example

```
\definetypevimfile[typeRUBY] [syntax=ruby]
```

after which one can use

```
\typeRUBY[option]{rubyfile}
```

```
\def\definetypevimfile
  {\dodoubleargument\dodefinetypevimfile}
```

```
\def\dodefinetypevimfile[#1][#2]%
  {\unexpanded\setvalue{#1}{\dodoubleempty\notypevimfile[#2]}}
```

\definevim.. This macro allows you to pretty print code snippets. For example

```
\definevimtyping [RUBY] [syntax=ruby]
\startRUBY
# This is my first ruby program
puts "Hello World"
\stopRUBY
```

gives

```
# This is my first ruby program
puts "Hello World"
```

```
\def\definevimtyping
  {\dodoubleargument\dodefinevimtyping}
```

```
\def\dodefinevimtyping[#1][#2]%
  {\setevalue{\e!start#1}{\noexpand\dostartbuffer[vimsyntax][\e!start#1][\e!stop#1]}%
   \setvalue{\e!stop#1}{\saveandtypevimfile[#2]}}
```

Some defaults.

```
\setupvimtyping
  [        syntax=context,
           \c!tab=8,
         \c!space=\v!yes,
         \c!start=1,
          \c!stop=0,
        \c!before=,
         \c!after=,
     \c!numbering=\v!off,
 \c!numbercommand=\inleft,
   \c!numberstyle=\v!smallslanted,
   \c!numbercolor=,
          \c!step=1,
      colorscheme=default,
  ]
```

Pre-defined Syntax : This is based on `ps_color.vim`, which does not use any bold typeface.

VIM uses hex mode for setting colors, I do not want to convert them to rgb values.

```
\startvimcolorscheme[default]
```

```
\setupcolor[hex]

\definecolor  [vimsyntax!default!Special]     [h=907000]
\definecolor  [vimsyntax!default!Comment]     [h=606000]
\definecolor  [vimsyntax!default!Number]      [h=907000]
\definecolor  [vimsyntax!default!Constant]    [h=007068]
\definecolor  [vimsyntax!default!PreProc]     [h=009030]
\definecolor  [vimsyntax!default!Statement]   [h=2060a8]
\definecolor  [vimsyntax!default!Type]        [h=0850a0]
\definecolor  [vimsyntax!default!Todo]        [h=e0e090]

\definecolor  [vimsyntax!default!Error]       [h=c03000]
\definecolor  [vimsyntax!default!Identifier]  [h=a030a0]
\definecolor  [vimsyntax!default!SpecialKey]  [h=1050a0]
\definecolor  [vimsyntax!default!Underline]   [h=6a5acd]

\definevimsyntax
  [Normal]
  [\c!style=\tttf,\c!color=\maintextcolor]

\definevimsyntax
  [Constant]
  [\c!style=\v!mono,\c!color=vimsyntax!default!Constant]

\definevimsyntaxsynonyms
  [Character,Boolean,Float]
  [Constant]

\definevimsyntax
  [Number]
  [\c!style=\v!mono,\c!color=vimsyntax!default!Number]

\definevimsyntax
  [Identifier]
  [\c!style=\v!mono,\c!color=vimsyntax!default!Identifier]

\definevimsyntaxsynonyms
  [Function]
  [Identifier]

\definevimsyntax
  [Statement]
  [\c!style=\v!mono,\c!color=vimsyntax!default!Statement]

\definevimsyntaxsynonyms
  [Conditional,Repeat,Label,Operator,Keyword,Exception]
  [Statement]

\definevimsyntax
  [PreProc]
  [\c!style=\v!mono,\c!color=vimsyntax!default!PreProc]

\definevimsyntaxsynonyms
  [Include,Define,Macro,PreCondit]
  [PreProc]

\definevimsyntax
  [Type,StorageClass, Structure, Typedef]
  [\c!style=\v!mono, \c!color=vimsyntax!default!Type]

\definevimsyntax
  [Special]
  [\c!style=\v!mono,\c!color=vimsyntax!default!Special]

\definevimsyntax
  [SpecialKey]
  [\c!style=\v!mono,\c!color=vimsyntax!default!SpecialKey]

\definevimsyntax
  [Tag,Delimiter]
  [\c!style=\v!mono]

\definevimsyntax
  [Comment,SpecialComment]
  [\c!style=\v!mono,\c!color=vimsyntax!default!Comment]

\definevimsyntax
  [Debug]
  [\c!style=\v!mono]

\definevimsyntax
  [Underlined]
  [\c!style=\v!mono,\c!command=\underbar]

\definevimsyntax
  [Ignore]
  [\c!style=\v!mono]

\definevimsyntax
  [Error]
  [\c!style=\v!mono,\c!color=vimsyntax!default!Error]
```

```
\definevimsyntax
  [Todo]
  [\c!style=\v!mono,\c!color=vimsyntax!default!Todo]

\stopvimcolorscheme

\startvimcolorscheme[blackandwhite]

\definevimsyntax
  [Normal]
  [\c!style=\tttf,\c!color=\maintextcolor]

\definevimsyntax
  [Constant]
  [\c!style=\v!mono,\c!color=]

\definevimsyntaxsynonyms
  [Character,Boolean,Float]
  [Constant]

\definevimsyntax
  [Number]
  [\c!style=\v!mono,\c!color=]

\definevimsyntax
  [Identifier]
  [\c!style=\v!mono,\c!color=]

\definevimsyntaxsynonyms
  [Function]
  [Identifier]

\definevimsyntax
  [Statement]
  [\c!style=\v!mono\v!bold,\c!color=]

\definevimsyntaxsynonyms
  [Conditional,Repeat,Label,Operator,Keyword,Exception]
  [Statement]

\definevimsyntax
  [PreProc]
  [\c!style=\v!bold\v!mono,\c!color=]
```

```
\definevimsyntaxsynonyms
  [Include,Define,Macro,PreCondit]
  [PreProc]

\definevimsyntax
  [Type,StorageClass, Structure, Typedef]
  [\c!style=\v!bold\v!mono, \c!color=]

\definevimsyntax
  [Special]
  [\c!style=\v!mono,\c!color=]

\definevimsyntax
  [SpecialKey]
  [\c!style=\v!mono,\c!color=]

\definevimsyntax
  [Tag,Delimiter]
  [\c!style=\v!mono,\c!color=]

\definevimsyntax
  [Comment,SpecialComment]
  [\c!style=\v!slanted\v!mono,\c!color=]

\definevimsyntax
  [Debug]
  [\c!style=\v!mono,\c!color=]

\definevimsyntax
  [Underlined]
  [\c!style=\v!mono,\c!color=,\c!command=\underbar]

\definevimsyntax
  [Ignore]
  [\c!style=\v!mono,\c!color=]

\definevimsyntax
  [Error]
  [\c!style=\v!mono,\c!color=,\c!command=\overstrike]

\definevimsyntax
  [Todo]
  [\c!style=\v!mono,\c!command=\inframed]
```

```
\stopvimcolorscheme

\protect

\stopmodule
```

An example usage:

```
\doifnotmode{demo}{\endinput}

\setupcolors[state=start]

\usetypescript[modern-base][texnansi]

\setupbodyfont[modern,10pt]

\starttext

\title{Matlab Code Listing -- Color}

\definevimtyping  [MATLAB]  [syntax=matlab]

\startMATLAB
function russell_demo()
r = 3; c = 4; p = 0.8; action_cost = -1/25;
obstacle = zeros(r,c); obstacle(2,2)=1;
terminal = zeros(r,c); terminal(1,4)=1; terminal(2,4)=1;
absorb = 1;
wrap_around = 0;
noop = 0;
T = mk_grid_world(r, c, p, obstacle, terminal, absorb, wrap_around, noop);
nstates = r*c + 1;
if noop
  nact = 5;
else
  nact = 4;
end
R = action_cost*ones(nstates, nact);
R(10,:)  = 1;
R(11,:)  = -1;
R(nstates,:)  = 0;
discount_factor = 1;

V = value_iteration(T, R, discount_factor);
```

```
Q = Q_from_V(V, T, R, discount_factor);
[V, p] = max(Q, [], 2);

use_val_iter = 1;
[p,V] = policy_iteration(T, R, discount_factor, use_val_iter);

\stopMATLAB

\title{Lua Code Listing -- Black and White}

\definevimtyping  [LUA]  [syntax=lua,colorscheme=blackandwhite]

\startLUA
-- version   :  1.0.0 - 07/2005
-- author    :  Hans Hagen - PRAGMA ADE - www.pragma-ade.com
-- copyright :  public domain or whatever suits
-- remark    :  part of the context distribution

-- TODO: name space for local functions

-- loading:  scite-ctx.properties

-- generic functions

local crlf = "\n"

function traceln(str)
    trace(str ..  crlf)
    io.flush()
end

table.len  = table.getn
table.join = table.concat

function table.found(tab, str)
    local l, r, p
    if string.len(str) == 0 then
        return false
    else
        l, r = 1, table.len(tab)
        while l <= r do
            p = math.floor((l+r)/2)
            if str < tab[p] then
                r = p - 1
```

```lua
            elseif str > tab[p] then
                l = p + 1
            else
                return true
            end
        end
        return false
    end
end

function string.grab(str, delimiter)
    local list = {}
    for snippet in string.gfind(str,delimiter) do
        table.insert(list, snippet)
    end
    return list
end

function string.join(list, delimiter)
    local size, str = table.len(list), ''
    if size > 0 then
        str = list[1]
        for i = 2, size, 1 do
            str = str ..  delimiter ..  list[i]
        end
    end
    return str
end

function string.spacy(str)
    if string.find(str,"^%s*$") then
        return true
    else
        return false
    end
end

function string.alphacmp(a,b,i) -- slow but ok
    if i and i > 0 then
        return string.lower(string.gsub(string.sub(a,i),'0',' ')) <
string.lower(string.gsub(string.sub(b,i),'0',' '))
    else
        return string.lower(a) < string.lower(b)
    end
end

function table.alphasort(list,i)
    table.sort(list, function(a,b) return string.alphacmp(a,b,i) end)
end
```

```
function io.exists(filename)
    local ok, result, message = pcall(io.open,filename)
    if result then
        io.close(result)
        return true
    else
        return false
    end
end

function os.envvar(str)
    if os.getenv(str) ˜= '' then
        return os.getenv(str)
    elseif os.getenv(string.upper(str)) ˜= '' then
        return os.getenv(string.upper(str))
    elseif os.getenv(string.lower(str)) ˜= '' then
        return os.getenv(string.lower(str))
    else
        return ''
    end
end

function string.expand(str)
    return string.gsub(str, "ENV%((%w+)%)", os.envvar)
end

function string.strip(str)
    return string.gsub(string.gsub(str,"^%s+",''),"%s+$",'')
end

function string.replace(original,pattern,replacement)
    local str = string.gsub(original,pattern,replacement)
--     print(str) -- indirect, since else str + nofsubs
    return str -- indirect, since else str + nofsubs
end

\stopLUA

\stoptext
```