Zpravodaj Československého sdružení uživatelů TeXu

André Simon

The Highlight Programme: Code and Syntax Highlighting

Zpravodaj Československého sdružení uživatelů TeXu, Vol. 19 (2009), No. 4, 222–239

Persistent URL: http://dml.cz/dmlcz/150100

Program **Highlight** a jeho užití

André Simon

Contents

Abstrakt

Článek představuje program Highlight, `http://andre-simon.de/`, který je určen k formátování zdrojových kódů. Průřezově představuje jeho možnosti a potenciál. Program umí pracovat v grafickém režimu, dávkově a může být načten jako externí knihovna.

Více či méně použitelných alternativ je relativně hodně. Namátkou zmiňme TEX balíčky fancyvrb a listings. Mimo TEX pak:

- GeSHi – Generic Syntax Highlighter, `http://qbnz.com/highlighter/`,
- GNU Source-highlight, `http://www.gnu.org/software/src-highlite/`, či,
- Pygments, `http://pygments.org/`.

V rámci experimentů představuje autor ve svém programu přístup k formátování více jazyků v jednom dokumentu, např. u dokumentů a webových stránek při kombinaci HTML+PHP+CSS+JavaScript, LATEX+Sweave+R, Perl+TEX, Lua+TEX apod.

Klíčová slova: Program Highlight, zvýrazňování syntaxe, formátování zdrojových kódů s více jazyky.

1. Introduction

Syntax highlighting is a must-have feature for a programmer's text editor. The colouring of language elements helps to quickly overview large code portions mingled with comments, or to correct syntax errors before the compilation process is started. Highlight is a utility to preserve code highlighting when the source code is published on the web or in print documents. Apart from adding colours to the output, it contains reformatting features to obtain a consistent code layout.

2. Features

Highlight takes an input file and searches for syntax elements which should be outputted in a distinct format. These elements may be keywords, comments, strings, numbers, escape sequences and directives. Syntax elements are stored in text configuration files (language definitions). The accompanying formatting information is stored in separate configuration files (colour themes). These themes define font faces, language element colours and background colour. Both configuration entities may be customized by the user. Highlight is delivered with 140 language definitions and 40 colour themes.

The following output file types are supported: HTML, XHTML 1.1, RTF, TEX, LATEX, SVG, BBCode, terminal escape sequences and XML. Most of the formats have special options to reduce manual editing of the result.

Long input lines may be wrapped automatically, taking care of function call and statement indentation. C-style languages can be reformatted using several indentation styles (including GNU and K&R variants).

If the output format supports style files, the formatting information is stored in referenced files by default (HTML and SVG: CSS files, TEX and LATEX: `sty` files). The separation of styles and content is more efficient and simplifies design changes of existing documents. Style files may include user defined files to override and enhance highlight's default formatting.

Highlight is distributed as library, command line (CLI) and graphical user interface (GUI). The CLI offers the complete feature set, whereas the GUI includes an instant preview to show the impact of formatting changes.

3. Command line and graphical interfaces

The following examples show how to produce a highlighted C++ file, using an input file called `main.cpp`:

- Generate HTML:

  ```
  highlight -i main.cpp -o main.cpp.html
  highlight < main.cpp > main.cpp.html --syntax cpp
  ```

  You will find the HTML file `main.cpp.html` and `highlight.css` in the working directory. If no input filename is defined by `--input` or given at the prompt, highlight is not able to determine the language type by means of the file extension. Only some scripting languages are determined by the shebang in the first input line. In this case you have to pass Highlight the given language with `--syntax` (this should be the file suffix of the source file in most cases).

- Generate LATEX with embedded style definitions and line numbers:

  ```
  highlight --latex -i main.cpp -o main.cpp.tex
     --include-style --linenumbers
  ```

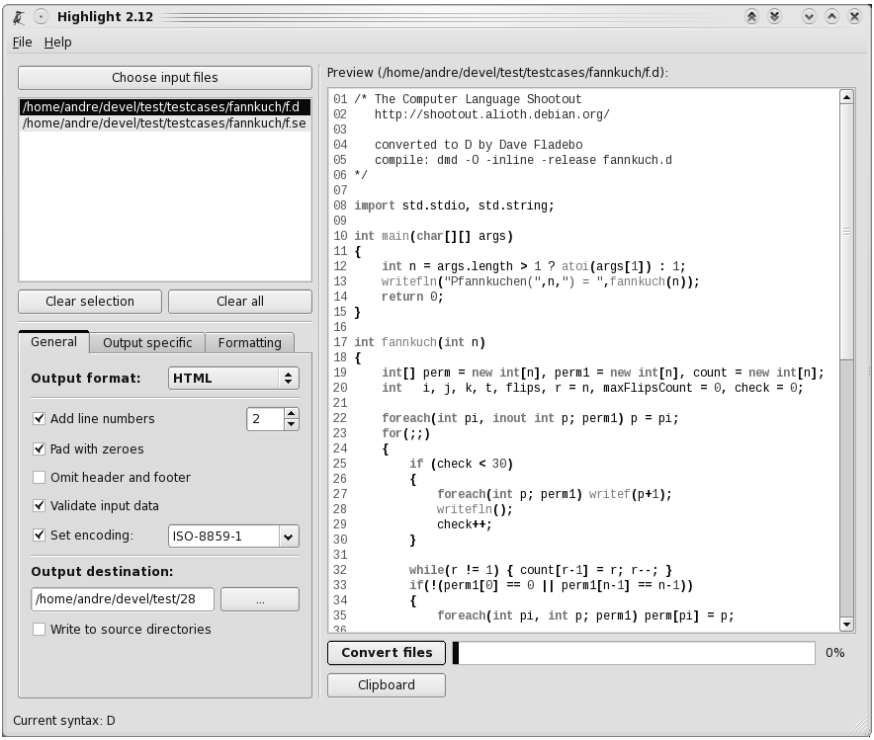
- Generate plain TEX using "ANSI" source formatting style and "print" colour theme:

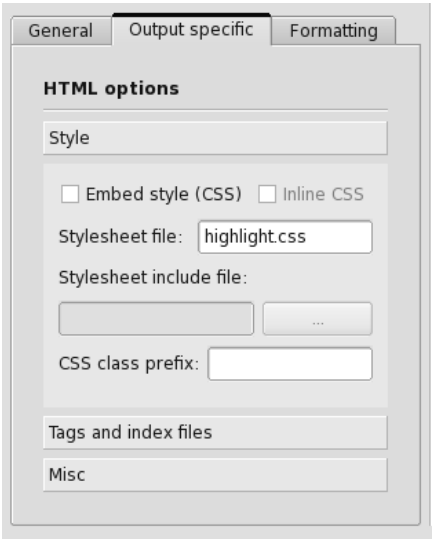
  ```
  highlight --tex -i main.cpp -o main.cpp.tex
     --reformat ansi --style print
  ```

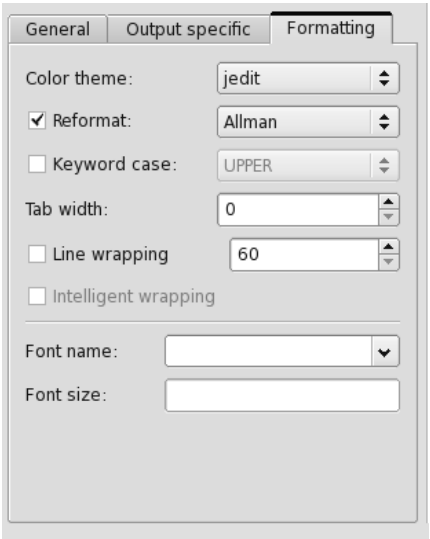
- Define an output directory:

  ```
  highlight -O some/target/dir/ *.cpp *.h
  ```

Screenshots on the next page show the preview window and the option panes.


Highlight 2.12
File Help
Choose input files
/home/andre/devel/test/testcases/fannkuch/f.d
/home/andre/devel/test/testcases/fannkuch/f.se
Clear selection
Clear all
General
Output specific
Formatting
Output format:
HTML
Add line numbers
2
Pad with zeroes
Omit header and footer
Validate input data
Set encoding:
ISO-8859-1
Output destination:
/home/andre/devel/test/28
...
Write to source directories
Preview (/home/andre/devel/test/testcases/fannkuch/f.d):
01 /* The Computer Language Shootout
02 http://shootout.alioth.debian.org/
03
04 converted to D by Dave Fladebo
05 compile: dmd -O -inline -release fannkuch.d
06 */
07
08 import std.stdio, std.string;
09
10 int main(char[][] args)
11 {
12 int n = args.length > 1 ? atoi(args[1]) : 1;
13 writefln("Pfannkuchen(",n,") = ",fannkuch(n));
14 return 0;
15 }
16
17 int fannkuch(int n)
18 {
19 int[] perm = new int[n], perm1 = new int[n], count = new int[n];
20 int i, j, k, t, flips, r = n, maxFlipsCount = 0, check = 0;
21
22 foreach(int pi, inout int p; perm1) p = pi;
23 for(;;)
24 {
25 if (check < 30)
26 {
27 foreach(int p; perm1) writef(p+1);
28 writefln();
29 check++;
30 }
31
32 while(r != 1) { count[r-1] = r; r--; }
33 if(!(perm1[0] == 0 || perm1[n-1] == n-1))
34 {
35 foreach(int pi, int p; perm1) perm[pi] = p;
Convert files
0%
Clipboard
Current syntax: D


General
Output specific
Formatting
HTML options
Style
Embed style (CSS)
Inline CSS
Stylesheet file:
highlight.css
Stylesheet include file:
...
CSS class prefix:
Tags and index files
Misc


General
Output specific
Formatting
Color theme:
jedit
Reformat:
Allman
Keyword case:
UPPER
Tab width:
0
Line wrapping
60
Intelligent wrapping
Font name:
Font size:

4. Output options

The following incomplete list shows the most interesting options applicable to all output formats:

--style-outfile=<file>, --style-infile=<file>
Define style definition filename, and the file to be included in style-outfile.

--fragment
Omit header and footer of the output to embed it in existing documents.

--reformat=<style>
Reformat output in given style, where <style> is one of allman, banner, gnu, java, k&r, linux, stroustrup, or whitesmith. This feature is available for C-style syntax.

--include-style
Include style definition in each output file.

--wrap, --wrap-simple, --line-length=<num>
Wrap long lines with or without taking care of function call indentation. Set line length before wrapping.

--font=<font>, --font-size=<size>
Set font face and size (specific to output format).

--linenumbers, --line-number-start=<cnt>,
--line-number-length=<num>
Print line numbers, set line numbering start and length incl. left padding.

--style=<style name> Set highlighting colour style.

--replace-tabs=<num> Replace tabs by given number of spaces.

--encoding=<enc>
Set output encoding which matches input file encoding; omit encoding information if <enc> is "NONE".

--kw-case=<upper|lower|capitalize>
Output all keywords in given case if language is not case sensitive.

--start-nested=<language>
Define nested language which starts input without opening delimiter.

4.1. HTML/XHTML options

--anchors Attach anchors to line numbers (HTML only).

--anchor-prefix=<prefix> Set anchor name prefix.

--anchor-filename Use input file name as anchor name.

--print-index Output index file with links to all output files.

--ordered-list Output lines as ordered list items (assumes `--linenumbers`).

--class-name=<name> Set CSS class name to avoid name clashes.

--inline-css
Output CSS information within each tag (generates verbose output).

--mark-line='n[=txt]; m'
Mark given lines n..m and add optional help texts as tooltips.
--enclose-pre Enclose fragmented output in pre tags (assumes `--fragment`).
--ctags-file[=<file>] Read ctags file to include meta information as tooltips (default file value: "tags").

4.2. LaTeX options

--babel Disable babel package shorthands.
--replace-quotes Replace double quotes by \dq.
--pretty-symbols Add character boxes to improve appearance of brackets and other symbols.

The resulting LaTeX articles require the color package. If the output is UTF-8 encoded, the ucs package is needed. If the input lines are wrapped, the marvosym package is included to represent inserted line breaks by the right torque symbol.

The formatting information is stored as commands with a "hl" prefix. These commands are used to enclose recognized syntax tokens in the output. Whitespace is escaped as "\ " to assure correct indentation of code blocks. The `--babel` and `--replace-quotes` options can be applied to gain compatiblity with babel or other packages. The `--pretty-symbols` option adds boxes for special characters like parantheses, @, # etc. These boxes improve the appearance of the characters by fitting their dimensions to the surrounding text.

Plain TeX documents include macros to apply formatting to recognized syntax tokens. The macro naming scheme corresponds to the LaTeX output. A "special" command is used to set the background colour, which may not be supported by all `dvi` drivers. Page numbering is disabled.

4.3. RTF options

--page-size=<size>
Set page size, where <size> is one of a3, a4, a5, b4, b5, b6, or letter.
--char-styles Include character stylesheets.

4.4. SVG options

--height=<height>, --width=<width>
Set image height and width (units allowed).

5. Configuration

5.1. File format

All Highlight configuration files are stored as plain ASCII text files, using the convention *$ParamName=ParamValue*.

Parameter names are not case sensitive. The value may be a single character, a list of words or a regular expression. Lists may be split in multiple lines. Comments start with # as the first character of a line.

5.2. Language definitions

A language definition describes all elements of a programming language which will be highlighted by different colours and font types.

File format:

```
# Regular expression to describe valid number tokens.
# Default value is set in the parser.
$DIGIT=regex(<RE>)

# Regular expression to describe valid identifier tokens.
# Default value is set in the parser.
$IDENTIFIER=regex(<RE>)

# List of keywords or regular expressions.
# <group> is the name of the keyword group.
# The group must be defined in the applied colour theme to provide a matching
# highlighting style.
$KEYWORDS(<group>)=regex(<RE> <, GROUP-NUM>) | <List>

# List of String delimiters.
$STRINGDELIMITERS=<List>

# List of string delimiters which are not equal (open != close).
$STRING_UNEQUAL=<open close>

# List of escape characters in Strings (ie. "\") or regular expression.
$ESCCHAR=<List> | regex(<RE>)

# Escape characters may appear outside of strings.
$ALLOWEXTESCAPE=<true|false>

# Prefix which disables highlighting of escape characters within a string.
$RAWSTRINGPREFIX=<character>

# Delimiters of multi line comments.
# Delimiter comment_close may be emitted if $ALLOWNESTEDCOMMENTS is false.
$ML_COMMENT=<comment_begin comment_close>

# List of strings which start single line comments.
$SL_COMMENT=<List> | regex(<RE>)

# Prefix of preprocessor directive lines.
$DIRECTIVE=<prefix> | regex(<RE>)

# Character which continues a compiler directive after a line break.
$CONTINUATIONSYMBOL=<symbol>

# Source code may be reformatted (only C-style languages!).
$REFORMATTING=<true | false>

# Symbols (brackets or operators).
$SYMBOLS=<List>
```

```
# Multiple line comments may be nested.
$ALLOWNESTEDCOMMENTS=<true | false>

# Programming language is case sensitive.
$IGNORECASE=<true | false>

# Include another language definition stored in the same data directory.
$INCLUDE=<language definition>

# Define the opening and closing delimiter expressions of the embedded language.
# There may be multiple entries for the same language.
$NESTED(language)=regex(<RE>) regex(<RE>)
```

5.3. Regular expressions

Highlight supports Perl-style regular expressions, which are evaluated for each input line. They are defined as *regex(<RE> <, GROUP-NUM>)*, where RE is the regex string, and GROUP-NUM is an optional parameter which defines the matching group number. If the regex contains multiple grouping parantheses, GROUP-NUM is the number of the group, whose match should be returned as a highlighted element (count number from left to right). The capturing states of the groups are irrelevant for counting. Legal values: 0 <= GROUP-NUM <= highest group index. A GROUP-NUM of 0 describes the complete match. If GROUP-NUM is undefined, the group match with the highest number will be returned.

Regex Examples:

```
$KEYWORDS(kwa)=regex([A-Z]\w+)
```

Highlight identifiers beginning with a capital letter.

```
1. $KEYWORDS(kwc)=regex(\$\{(\w+)\}), or
2. $KEYWORDS(kwc)=regex(\$\{(\w+)\}, 1)
```

Highlight variable names like $name. Only the name is highlighted as a keyword (\w+). The grouping feature is used to achieve this effect. If no capturing group index is defined, the right-most group's match (highest capturing index) is returned, see Example 1 above.

```
$KEYWORDS(kwd)=regex((\w+)\s*\()
```

Highlight method names. Note that grouping is used again.

5.4. Nested syntax configuration

Language definitions may contain nested language delimiters to change the syntax information while parsing a file. This is useful to highlight mixed code files like HTML with PHP instructions. It is also applicable to literate programming, inputs like LuaTEX and Sweave files. Currently the following code combinations are configured:

- Pascal + Assembly.
- HTML + PHP + CSS + JavaScript.
- LaTeX + Perl + Lua + R.

If the file starts with the nested syntax omitting the open delimiter, the `--start-nested` option will load the given syntax, see LuaTeX output examples.

5.4.1. Pascal definition file

```
$DESCRIPTION=Pascal
$KEYWORDS(kwa)= absolute abstract and array as asm assembler
   automated begin [...]
$KEYWORDS(kwb)=boolean char integer pointer real text true
   false cardinal [...]
$KEYWORDS(kwc)=if else then downto do for repeat while
   to until with
$KEYWORDS(kwd)=regex((\w+)\s*\()
$STRINGDELIMITERS=" '
$SL_COMMENT=//
$ML_COMMENT={ } (* *)
$IGNORECASE=true
$SYMBOLS= ( ) [ ] , ; : & | < > !  = / * %  + - @ . ^
$ESCCHAR=regex(\#\$\p{XDigit}{2}|\#\d{,3})
$DIGIT=regex((?:0x|0X|\$)[0-9a-fA-F]+|\d*[\.\_]?\d+(?:[eE]
   [\-\+]\d+)?[lLuUbfdm]*)
$ALLOWEXTESCAPE=true
$NESTED(asm)=regex(asm) regex(end;)
```

5.4.2. HTML definition file

```
$DESCRIPTION=HTML
   # [...]
$NESTED(php)=regex(\<\?php) regex(\?\>)
$NESTED(jsp)=regex(\<\%[@!\=]?) regex(.*(%>).*)
$NESTED(css)=regex(\<style\s+type\=\"text\/css\"\>)regex(\<\/style\>)
$NESTED(js)=regex(\<script\s+language\=\"JavaScript\"(?:\s+
   type\=\"text\/javascript\")?\>)regex(\<\/script\>)
```

5.4.3. LaTeX definition file

```
$DESCRIPTION=TeX and LaTeX
   # [...]
# PerlTeX delimiters
# Issue: } is also a Perl symbol
$NESTED(pl)=regex(\\perl(?:re)?newcommand\{\\\w+\}\[.*\]\{)
   regex((?<!\{)\})
```

```
$NESTED(pl)=regex(\\perl(?:re)?newenvironment\{\w+\}\[.*\]\{)
   regex((?<!\{)\})

# LuaTeX delimiters
# Issue: } is also a Lua symbol
$NESTED(lua)=regex(\\directlua.*?\{) regex((?<!\{)\})

# Sweave delimiters
$NESTED(r)=regex(^\<\<.*\>\>\=) regex(@)
```

5.4.4. Example: LaTeX, Sweave and R

Formatted example by Professor Leisch follows, see Example 3 (`http://www.stat.uni-muenchen.de/~leisch/Sweave/example-3.Snw`).

documentclass[a4paper]{article}
begin{document}
<<**echo=false,results=hide**>>=
library(**lattice**)
library(**xtable**)
data(cats, package=*"MASS"*)
@
section*{The Cats Data}
Consider the **texttt**{cats} regression example from Venables \& Ripley (1997). The data frame contains measurements of heart and body weight of **Sexpr**{nrow(cats)} cats (**Sexpr**{sum(cats$Sex=="F")} female, **Sexpr**{sum(cats$Sex=="M")} male).
A linear regression model of heart weight by sex and gender can be fitted in R using the command
<<>>=
lm1 = **lm**(Hwt∼Bwt*Sex, **data**=cats)
lm1
@
Tests for significance of the coefficients are shown in Table∼**ref**{tab:coef}, a scatter plot including the regression lines is shown in Figure∼**ref**{fig:cats}.
SweaveOpts{echo=false}
<<**results=tex**>>=
xtable(lm1, caption=*"Linear regression model for cats data."* , label=*"tab:coef"*)
@
begin{figure}
  centering
<<**fig=TRUE,width=12,height=6**>>=

```
lset(col.whitebg())
print(xyplot(Hwt~Bwt|Sex, data=cats, type=c("p", "r")))
@
  \caption{The cats data from package MASS.}
  \label{fig:cats}
\end{figure}
\begin{center}
\end{center}
\end{document}
```

5.4.5. Example: First LuaTEX output

Formatted source code from the [NTG-context][Dev-luatex] mailing list, see `http://markmail.org/message/h3d6ju5c5umah57h`.

```
(* LuaTeX test *)
\def\sortedsequence#1#2{\directlua {
      do
          local sep = "\luaescapestring{#1}"
          local str = "\luaescapestring{#2}"
          local tab = {}
          for s in string.gmatch(str,"([^"..sep.."]+)") do
              tab[\string ##tab+1] = s
          end
          table.sort(tab)
          tex.sprint(table.concat(tab,sep))
      end
}}
\expandafter\ifx\csname directlua\endcsname \relax \else
\directlua 0 {tex.enableprimitives('',tex.extraprimitives ())}
\fi
(* End *)
```

5.4.6. Example: Second LuaTEX output

A LuaTEX output example follows, for the full source code you may like to check `http://www.andre-simon.de/dokuwiki/doku.php?id=luatex` and the website at `http://lua-users.org/wiki/MakingLuaLikePhp`.

```
% This LuaTeX file starts with Lua code: to highlight it properly use this cmd:
% highlight --latex --start-nested=inc_luatex input.tex
function explode(str,divider)
 local result = {}
```

```
 % Function body omitted for brevity.
 return result
end

function implode(t,div)
 return table.concat(t,div)
end

% #1 = continuation (called with the sorted list as its only
% parameter), #2 = delimiter, #3 = <delimiter>-separated list
\def\sort#1#2#3{%
 \directlua0 {%
  % We convert the string into a table, delimited by a
  % (unexpanded, escaped) delimiter:
  local t = explode("\luaescapestring{\unexpanded{#3}}",
       "\luaescapestring{\unexpanded{#2}}")
  % We sort the table in-place:
  table.sort(t)
  % We output the table as a string.
  tex.print("\luaescapestring{\unexpanded{#1}}{" ..
   implode(t," ") .. "}")
 }
}

% Output tokens without expansion.
\def\typeout#1{\message{\unexpanded{[#1]}}}

\sort{\typeout}{ }{%
This file demonstrates LuaTeX.}
\bye
```

5.5. Colour themes

Colour themes contain the formatting information of the language elements which are described in language definitions.

The themes have to be stored as `*.style` files in the themes directory.

File format:

```
<Colour> = #RRGGBB
RR GG BB describe the red/green/blue hex-values which define the colour.
Value range: 00 (none) - FF (full)

<Format> = <bold> <italic> <underline>
Bold, italic and underline are optional attributes and may be combined.
```

```
# Colour of unspecified text
$DEFAULTCOLOUR=<Colour>

# Background colour
$BGCOLOUR=<Colour>

# Font size
$FONTSIZE=<number>

# Formatting of keywords, which belong to the corresponding keyword group
$KW-GROUP(<group>)=<Colour> <Format>

# Formatting of numbers
$NUMBER=<Colour> <Format>

# Formatting of escape characters
$ESCAPECHAR=<Colour> <Format>

# Formatting of strings
$STRING=<Colour> <Format>

# Formatting of comments
$COMMENT=<Colour> <Format>

# Formatting of single line comm. (optional, equals to $COMMENT if omitted)
$SL-COMMENT=<Colour> <Format>

# Formatting of compiler directives
$DIRECTIVE=<Colour> <Format>

# Formatting of strings within compiler directives
$STRING-DIRECTIVE=<Colour> <Format>

# Formatting of symbols (optional, equals to $DEFAULTCOLOUR if omitted)
$SYMBOL=<Colour> <Format>

# Formatting of line numbers
$LINE=<Colour> <Format>

# Background colour of marked lines
$MARK-LINE=<Colour>
```

Example of Darkblue:

```
$DEFAULTCOLOUR=#ffffff
$BGCOLOUR=#000040
$FONTSIZE=10
$KW-GROUP(kwa)=#e2e825
$KW-GROUP(kwb)=#60ff60
$KW-GROUP(kwc)=#26e0e7
$KW-GROUP(kwd)=#e825e2 bold
$NUMBER=#42cad9
$ESCAPECHAR=#a1a1ff
$STRING=#ffa0a0
$STRING-DIRECTIVE=#ffa0a0
$COMMENT=#80a0ff
$SL-COMMENT=#ff7f9f
$DIRECTIVE=#ff80ff
$LINE=#e5d28e
$SYMBOL=#bababa
$MARK-LINE=#006600
```

5.6. Keyword groups

You may define custom keyword groups and corresponding highlighting styles. This is useful if you want to highlight functions of a third party library, macros, constants etc. You define a new group in two steps:

- Define a group in your language definition:
  $KEYWORDS(group)
  The group attribute is the name of the new keyword group. You may use the same group name for multiple entries.
- Add a corresponding highlighting style in your colour theme:
  $KW-GROUP(group) = #RRGGBB <bold> <italic> <underline>

Note that every group name which is listed in a language definition should be defined in the used colour theme. The keyword groups "kwa"–"kwd" are predefined in all packaged colour themes.

```
# Some language definition...
$KEYWORDS(kwa)=for repeat while [...]
$KEYWORDS(debug)=ASSERT DEBUG
$ML_COMMENT=/* */
   # [...]

# Some colour theme...
$KW-GROUP(kwa)=#ddbb00 bold
$KW-GROUP(debug)=#ff0000 bold
$COMMENT=#978345 italic
   # [...]
```

5.7. File type mapping

Several programming languages make use of multiple file types to separate interface definitions from implementation modules, or there exists several file endings for historical reasons. Unix scripts usually contain the path of the script interpreter in the first comment line, which is called a shebang. The file type extensions and shebang patterns are mapped to language definitions using the `filetypes.conf` file:

```
# List of extensions
$ext(ada)=adb ads a gnad
$ext(ampl)=dat run
$ext(amtrix)=s4 s4t s4h hnd t4
$ext(asm)=a51 29k 68s 68x x68
   # [...]

# Input file recognition
# Highlight matches the first input line with the listed expressions.
# Format: $shebang(language) = <regex>
```

```
$shebang(sh)=^#!(\/usr)?(\/local)?\/bin\/(bash|t?csh|[akz]?sh)
$shebang(pl)=^#!(\/usr)?(\/local)?\/bin\/perl
$shebang(py)=^#!(\/usr)?(\/local)?\/bin\/python
$shebang(awk)=^#!(\/usr)?(\/local)?\/bin\/[gn]?awk
```

All file extensions listed in `filetypes.conf` may be used with the `--syntax` command line option.

6. Project milestones

The initial goal of Highlight was to provide a HTML generator which makes use of Cascading Style Sheets (CSS) instead of antiquated and clumsy font tags. Since a highlighted code includes a lot of formatted elements, CSS makes a big difference in terms of output file size and maintenance effort. Hence the utility was called "src2css", which stressed its purpose but was soon renamed to the more memorable "highlight".

Back at the beginning of the development in 2002, none of the available highlighting tools had configurable syntax and colour theme definitions: Either the syntax elements or the accompanying colour settings were hardcoded. To allow a flexible configuration without coding, Highlight's syntax descriptions were separated from colour themes, and all configuration files were stored as plain text format.

The initial HTML output generator code was soon generalized to provide more output formats: LaTeX, Plain TeX, RTF, terminal escape sequences, XML and BBCode followed. According to user feedback, LaTeX and TeX became the most demanded features besides HTML.

The Artistic Style library was included in 2003 to provide syntax reformatting and indentation of C-style languages (C, C++, C#, Java). This feature helped to achieve a consistent code layout, especially useful for print or presentations.

In 2005, regular expressions were added to syntax files which enabled the definition of complex patterns. This feature rendered hardcoded special cases within the parser unnecessary. A side effect of the regex integration was that the parser core had undergone very few changes since 2006, which reduced the testing effort of basic language parsing.

The first Highlight releases were tarballs with a command line interface. Later the GUI was developed using wxWidgets, and Highlight was distributed with both CLI and GUI builds. Later in 2009, wxWidgets was replaced by the Qt GUI toolkit.

While presenting this article in summer 2009 Pavel Stříž, of the Zpravodaj journal, thought about the possibility of letting Highlight parse input with nested source code, i.e. HTML+PHP, Lua+TeX, Perl+TeX, Python+TeX, R+LaTeX (Sweave), and related tools to assist in literate programming. However this was

not supported and nested language support was implemented in version 2.12 to complete the article.

7. Parser implementation

The *CodeGenerator* is the abstract base class of all output code generators. It contains the parser logic and offers the highlighter interface for clients. The inheriting generator classes have to implement several virtual methods to pass format specific information to the parser. First of all, the parser needs to know the document structure divided into header, body and footer. The code highlighting is performed with formatting tags. These opening and closing tags have to be defined before the parsing takes place. Each generator has to take care of character replacements: input characters have to be mapped to valid escape sequences (example: '>' becomes '>' in HTML).

The highlighting of syntax elements (tokens) is managed using states. A state is implemented as a method which outputs the current token in the corresponding format and decides on the next token of how to proceed. The state is finished when an ending delimiter token is recognized or if the input stream stops.

Starting in a root state, the parser branches to one of the following states:

- keyword
- number
- multi line comment
- single line comment
- string
- directive
- escape character
- symbol
- nested code begin
- nested code end
- end of line
- end of file
- whitespace

The *root state* outputs all text sections which are not recognized as tokens with default formatting. The *keyword state* handles reserved indentifiers or other tokens defined by regular expressions. This is the only state which results in multiple output formatting tags: Each keyword group comprises of its own formatting information. The *number state* is responsible for outputting numbers. The *comment states* handle single line and block comments, taking care of embedded comments. The other token-related states behave as their names suggest.

The *nested code states* process code sections of other programming languages enclosed in the main (host) language. When a nested section starts, the parser loads the assigned language definition to replace the host language syntax and stores the name of the previously loaded language. The ending delimiter expression (which is defined in the host language definition) is added to the new syntax information in order to recognize the end of the section. When the end of the section is reached, the syntax information of the host language is reloaded.

The remaining *whitespace states* have to be handled by every state, as they may reoccur. They are used to handle line numbering, the correct completion of opened tags and the parsing termination after the file ends.

Each state may branch to another state if appropriate: The string state may call the escape character state. The separation in state methods helps to control the parsing process because each highlighting element can be handled in an isolated and small code section.

8. Embedding Highlight

The Highlight parser is encapsulated in a static C++ library by default (a shared library makefile target is also available). All highlight functionality is obtained from the CodeGenerator base class, see the interface `codegenerator.h`. A CodeGenerator factory method returns an instance equivalent to the given output format, i.e. a LaTeXGenerator, which will transform a given input string or file into the chosen output format.

Apart from using the C++ library, you might prefer the SWIG tool which allows the integration of Highlight in over 10 programming languages. The Highlight distribution includes the SWIG interface file and makefiles for Python and Perl module compilation.

The following Perl code demonstrates the SWIG interface:

```
use highlight;
# get a generator instance (for HTML output)
my $gen = highlightc::CodeGenerator_getInstance($highlightc::HTML);
# initialize the generator with colour theme and syntax
$gen->initTheme("/usr/share/highlight/themes/kwrite.style");
$gen->loadLanguage("/usr/share/highlight/langDefs/c.lang");
# set some parameters
$gen->setIncludeStyle(1);
$gen->setEncoding("ISO-8859-1");
# get output string
my $output=$gen->generateString("int main(int argc, char **argv) {\n".
                                " HighlightApp app;\n".
                                " return app.run(argc, argv);\n".
                                "}\n");
print $output;
# clear the instance
highlightc::CodeGenerator_deleteInstance($gen);
```

9. Plugins and editor integration

Since the Highlight CLI supports IO redirection using the `--syntax` option, it may be easily invoked within shell scripts. The package includes plugins for popular web software (DokuWiki, MovableType, Serendipity and WordPress). To quickly integrate Highlight in other projects a sample invocation code is provided such as PHP, Perl and Python classes.

The tutorial at `http://www.lyx.org/` shows how to invoke Highlight in a convenient manner from within LyX, see `http://wiki.lyx.org/Examples/IncludeExternalProgramListing`.

The SVG output option is used in the Inkscape plugin available at `http://xico.freeshell.org/`, which includes code snippets within vector graphics.

The Highlight documentation Wiki [1] also includes instructions to integrate the CLI interface in CodeBlocks, `http://www.codeblocks.org/`, and the ancient DevC++ programme, `http://www.bloodshed.net/devcpp.html`. A native plugin for Notepad++, `http://notepad-plus.sourceforge.net/`, is available in the Download section [1].

References

[1] Simon, André. Highlight: code & syntax highlighting. [Program Highlight: zvýrazňování zdrojových kódů a syntaxe.] [on-line, cit. September 19, 2009] The programme and its DokuWiki are available from the author's websites: Homepage of the programme: `http://www.andre-simon.de/` Wiki documentation: `http://wiki.andre-simon.de/`

Summary: The Highlight Programme: Code & Syntax Highlighting

The article presents features and options of the Highlight programme which is capable of highlighting source codes of different programming languages. The experimental approach of nested syntax configuration is tested on files with HTML+CSS+PHP+JavaScript, Perl+TeX, LaTeX+Sweave+R and Lua+TeX.

Key words: The Highlight programme, Syntax highlighting, Source code formatting, Nested syntax configuration.

André Simon, as@andre-simon.de
Eurener Str. 53C, Trier, DE-54294 Germany