GNU Source-highlight

given a source file, produces a document with syntax highlighting.
for GNU Source-highlight Version 2.1.2

by Lorenzo Bettini

Short Contents

Table of Contents

1 Introduction

GNU Source-highlight, given a source file, produces a document with syntax highlighting. The colors and the styles can be specified (bold, italics, underline) by means of a configuration file, and some other options can be specified at the command line.

The program already recognizes many programming languages (e.g., C++, Java, Perl, etc.) and file formats (e.g., log files, ChangeLog, etc.), and some output formats (e.g., HTML, ANSI color escape sequences, LaTeX, etc.). Since version 2.0, it allows you to specify your own input source language via a simple syntax described later in this manual (Chapter 7 [Language Definitions], page 14). Since version 2.1, it allows you to specify your own output format language via a simple syntax described later in this manual (Chapter 8 [Output Language Definitions], page 21).

The complete list of languages (indeed, file extensions) natively supported by this version of Source-highlight (2.1.2), as reported by `--lang-list`, is the following:

```
Supported languages (file extensions)
and associated language definition files

C = cpp.lang
H = cpp.lang
bison = bison.lang
c = cpp.lang
caml = caml.lang
cc = cpp.lang
changelog = changelog.lang
cpp = cpp.lang
flex = flex.lang
fortran = fortran.lang
h = cpp.lang
hh = cpp.lang
hpp = cpp.lang
htm = html.lang
html = html.lang
java = java.lang
javascript = javascript.lang
js = javascript.lang
l = flex.lang
latex = latex.lang
lex = flex.lang
lgt = logtalk.lang
ll = flex.lang
log = syslog.lang
logtalk = logtalk.lang
lua = lua.lang
ml = caml.lang
mli = caml.lang
pas = pascal.lang
pascal = pascal.lang
perl = perl.lang
php = php3.lang
php3 = php3.lang
pl = prolog.lang
```

```
pm = perl.lang
prolog = prolog.lang
py = python.lang
python = python.lang
rb = ruby.lang
ruby = ruby.lang
sig = sml.lang
sml = sml.lang
syslog = syslog.lang
tex = latex.lang
y = bison.lang
yacc = bison.lang
yy = bison.lang
```

The complete list of output formats natively supported by this version of Source-highlight (2.1.2), as reported by `--outlang-list`, is the following:

```
Supported output languages
and associated language definition files

esc = esc.outlang
esc-doc = esc.outlang
html = html.outlang
html-css = css_common.outlang
html-css-doc = cssdoc.outlang
html-doc = htmldoc.outlang
latex = latex.outlang
latex-doc = latexdoc.outlang
latexcolor = latexcolor.outlang
latexcolor-doc = latexcolordoc.outlang
texinfo = texinfo.outlang
xhtml = xhtml.outlang
xhtml-css = xhtmlcss.outlang
xhtml-css-doc = xhtmldoc.outlang
xhtml-doc = xhtmldoc.outlang
```

The meaning of the suffixes `-doc`, `-css` and `-css-doc` is explained in Section 5.4 [Output Language map], page 10.

Please, keep in mind, that I haven't tested personally all these language definitions: I actually checked that the definition file is correct (with the command line option `--check-lang`, Chapter 6 [Invoking source-highlight], page 12), but I'm not sure their definition actually respects that language syntax (e.g., I've put up together some language definitions by searching for information in the Internet, but I've never programmed in that language). So, if you find that a language definition is not precise, please let me know. Moreover, if you have a program example in a language that's not included in the '`tests`' directory, please send it to me so that I can include it in the test suite.

2 Installation

See the file 'INSTALL' for detailed building and installation instructions; anyway if you're used to compiling Linux software that comes with sources you may simply follow the usual procedure, i.e. untar the file you downloaded in a directory and then:

```
cd <source code main directory>
./configure
make
make install
```

Note: unless you specify a different install directory by `--prefix` option of configure (e.g. `./configure --prefix=<your home>`), you must be root to run `make install`.

Files will be installed in the following directories:

`Executables`
`/prefix/bin`

`docs and samples`
`/prefix/share/doc/source-highlight`

`conf files`
`/prefix/share/source-highlight`

Default value for prefix is `/usr/local` but you may change it with `--prefix` option to configure.

NOTICE: Originally, instead of Source-highlight, there were two separate programs, namely *GNU java2html* and *GNU cpp2html*. There are two shell scripts with the same name that will be installed together with Source-highlight in order to facilitate the migration (however their use is not advised and it is deprecated).

2.1 Download

You can download it from GNU's ftp site: ftp://ftp.gnu.org/gnu/src-highlite or from one of its mirrors (see http://www.gnu.org/prep/ftp.html).

I do not distribute Windows binaries anymore; since, they can be easily built by using Cygnus C/C++ compiler, available at http://www.cygwin.com. However, if you don't feel like downloading such compiler, you can request such binaries directly to me, by e-mail (find my e-mail at my home page) and I can send them to you. An MS-Windows port of Source-highlight is available from http://gnuwin32.sourceforge.net.

Archives are digitally signed by me (Lorenzo Bettini) with GNU gpg (http://www.gnupg.org). My GPG public key can be found at my home page (http://www.lorenzobettini.it).

You can also get the patches, if they are available for a particular release (see below for patching from a previous version).

2.2 Anonymous CVS Access

This project's CVS repository can be checked out through anonymous (pserver) CVS with the following instruction set. When prompted for a password for anoncvs, simply press the Enter key.

```
cvs -d:pserver:anoncvs@subversions.gnu.org:/cvsroot/src-highlite login

cvs -z3 -d:pserver:anoncvs@subversions.gnu.org:/cvsroot/src-highlite \
  co src-highlite
```

Further instructions can be found at the address:

http://savannah.gnu.org/projects/src-highlite.

Please notice that this way you will get the latest development sources of Source-highlight, which may also be unstable. This solution is the best if you intend to correct/extend this program: you should send me patches against the latest cvs repository sources.

If, on the contrary, you want to get the sources of a given release, through cvs, say, e.g., version X.Y.Z, you must specify the tag `rel_X_Y_Z` when you run the cvs command or the `cvs update` command.

When you compile the sources that you get through the cvs repository, before running the `configure` and `make` commands, you should, at least the first time, run the command:

```
sh reconf
```

This will run the autotools commands in the correct order, and also copy possibly missing files. You should have installed recent versions of `automake` and `autoconf` in order for this to succeed. You will also need `flex` and `bison`.

NOTICE: This convention holds since release 2.1.

2.3 What you need to build source-highlight

Since version 2.0 Source-highlight relies on regular expressions as provided by boost (http://www.boost.org), so you need to install at least the regex library from boost. Most GNU/Linux distributions provide this library already in a compiled form.

Source-highlight has been developed under GNU/Linux, using gcc (C++), and bison (yacc) and flex (lex), and ported under Win32 with Cygnus C/C++compiler, available at http://www.cygwin.com. I used the excellent GNU Autoconf and GNU Automake. I also used Autotools (ftp://ftp.ugcs.caltech.edu/pub/elef/autotools) which creates a starting source tree (according to GNU standards) with autoconf, automake starting files. Finally I used *GNU gengetopt* (http://www.gnu.org/software/gengetopt), for command line parsing.

I started to use also *doublecpp* (http://www.lorenzobettini.it/software/doublecpp) that permits achieving dynamic overloading.

If you want to use a specific version of the Boost regex library, you can use the configure option `--with-boost-regex` to specify a particular suffix. For instance,

```
./configure --with-boost-regex=boost_regex-gcc-1_31
```

Actually, apart from the boost regex library, you don't need the other tools above to build source-highlight because I provide generated sources, unless you want to develop source-highlight.

However, if you obtained sources through CVS, you need some other tools, see Section 2.2 [Anonymous CVS Access], page 3.

2.4 Patching from a previous version

If you downloaded a patch, say '`source-highlight-1.3-1.3.1-patch.gz`' (i.e., the patch to go from version 1.3 to version 1.3.1), cd to the directory with sources from the previous version (source-highlight-1.3) and type:

```
gunzip -cd ../source-highlight-1.3-1.3.1.patch.gz | patch -p1
```

and restart the compilation process (if you had already run configure a simple make should do).

2.5 Using source-highlight with less

This was suggested by Konstantine Serebriany. The script '`src-hilite-lesspipe.sh`' will be installed together with source-highlight. You can use the following environment variables:

```
export LESSOPEN="| /path/to/src-hilite-lesspipe.sh %s"
export LESS=' -R '
```

This way, when you use less to browse a file, if it is a source file handled by source-highlight, it will be automatically highlighted.

2.6 Building .rpm

Christian W. Zuckschwerdt added support for building an .rpm and an .rpm.src. You can issue the following command

```
rpm -tb source-highlight-2.1.2.tar.gz
```

for building an .rpm with binaries and

```
rpm -ts source-highlight-2.1.2.tar.gz
```

for building an .rpm.src with sources.

2.7 Related Software and Links

Martin Gebert is also implementing a KDE interface to source-highlight programs (and he did a wonderful job!), and it is called ksrc2html; if you want to test it: http://murphy.netsolution-net.de.

CGI support was enabled thanks to Robert Wetzel; I haven't tested it personally yet, so you may ask him directly. Moreover he set up some examples at the page http://www.inf.tu-dresden.de/~rw8/java2.html. If you want to use source-highlight as a CGI program, you have to use the executable source-highlight-cgi. You can build such executable by issuing

```
make source-highlight-cgi
```

in the '`src`' directory.

Moreover there's also a Java version of java2html, you can find it at http://www.generationjava.com/projects/Java2Html.shtml.

3 Copying Conditions

GNU Source-highlight is free software; you are free to use, share and modify it under the terms of the GNU General Public License that accompanies this software (see ‘`COPYING`’).

GNU `source-highlight` was written and maintained by Lorenzo Bettini http://www.lorenzobettini.it.

4 Simple Usage

Here are some realistic examples of running `source-highlight`[1].

Source-highlight only does a lexical analysis of the source code, so the program source is assumed to be correct!

Here's how to run source-highlight (for this example we will use C/C++ input files, but this is valid also for other source-highlight input languages):

```
source-highlight --src-lang cpp --out-format html \
    --input <C++ file> \
    --output <html file> \
    --style-file <style file> \
    options
```

For input files, apart from the `-i (--input)` option and the standard input redirection, you can simply specify some files at the command line and also use regular expressions (for instance '`*.java`'). In this case the name for the output files will be formed using the name of the source file with a .<ext> appended, where <ext> is the extension chosen according to the output format specified (in this example it would be .html). The style file (Section 5.1 [Output format style], page 8) contains information on how to format specific language parts (e.g., keywords in blue and boldface, etc.).

If `STDOUT` string is passed as `-o (--output)` option, then the output is forced to the standard output anyway.

If `-s (--src-lang)` is not specified, the source language is inferred by the extension of the input file (this, of course, does not work with standard input redirection).

If `-f (--out-format)` is not specified, the output will be produced in HTML.

If `--style-file` is not specified, the '`default.style`', which is included in the distribution, will be used (see Section 5.1 [Output format style], page 8 for further information).

4.1 LaTeX output

When using LaTeX output format you can choose between monochromatic output (by using `-f latex`) or colored output (by using `-f latexcolor`). Both formats make use of the `longtable` package, so be sure that you have this package installed (it should be in all current distributions). When using colored output, you need the `color` package (again this should be present in your system). Of course, you are free to define your own LaTeX output format, see Chapter 8 [Output Language Definitions], page 21.

4.2 Texinfo output

When using the Texinfo output format, you may want to use a dedicated style file, '`texinfo.style`', which comes with the source-highlight distribution, with the option `--style-file`. For instance, the example in Chapter 9 [Examples], page 26 is formatted with this style file.

4.3 ANSI color escape sequences

If you're using this output format, for instance together with `less` (see Section 2.5 [Using source-highlight with less], page 5), you may want to use the '`esc.style`', which comes with the source-highlight distribution, with the option `--style-file`. This should result in a more pleasant coloring output.

[1] Command lines that are too long are split into multiple indented lines separated by a \. Of course these commands are to be given in one line only, anyway.

5 Configuration files

During execution, source-highlight needs some files where it finds directives on how to recognize the source language (if not specified explicitly with `--src-lang` or `--lang-def`), on which output format to use (if not specified explicitly with `--out-format` or `--outlang-def`), on how to format specific source elements (e.g., keywords, comments, etc.), and source and output language definitions. These files will be explained in the next sections.

If the directory for such files is not explicitly specified with the command line option `--data-dir`, these files are searched for in the following order:

- the current directory;
- the installation directory for conf files, see Chapter 2 [Installation], page 3 (please keep in mind that this directory is hard-coded into source-highlight during compilation).

If you want to be sure about which file is used during the execution, you can use the command line option `--verbose`.

5.1 Output format style

You must specify your options for syntax highlighting in the file '`default.style`'[1]. Here's the one that comes with this distribution:

```
keyword blue b ;  // for language keywords
type darkgreen ; // for basic types
string red f ; // for strings and chars
comment brown i ; // for comments
number purple ; // for literal numbers
preproc darkblue b ; // for preproc directives (e.g. #include, import)
symbol darkred ; // for simbols (e.g. <, >, +)
function black b; // for function calls and declarations
cbracket red; // for block brackets (e.g. {, })

// line numbers
linenum black f;

// Internet related
url blue u, f;

// other elements for ChangeLog and Log files
date blue b ;
time darkblue b ;
ip darkgreen ;
file darkblue b ;
name darkgreen ;

// for Prolog, Perl...
variable darkgreen ;

// explicit for Latex
italics darkgreen i;
bold darkgreen b;
```

[1] Before version 2.1, this file was called '`tags.j2h`' which used to be a very obscure name. I hope this name convention is a better one :-).

```
underline darkgreen u;
fixed green f;
argument darkgreen;
optionalargument purple;
math orange;
```

You can specify your own file (it doesn't have to be named '`default.style`') with the command line option `--style-file`[2], see Chapter 6 [Invoking source-highlight], page 12.

You can also specify the color of normal text by adding this line

```
normal darkblue ;
```

As you might see the syntax of this file is quite straightforward:

```
b = bold
i = italics
u = underline
f = fixed
nf = not fixed
```

You may also specify more than on of these options separated by commas, e.g.

```
keyword blue u, b ;
```

Please keep in mind that in this case the order of these specified options is kept during the generation of the output; for instance, depending on the specific output format, the sequences `u, b` and `b, u` may lead to different results. In particular, the style that comes first is used after the ones that follow. For instance, in the case of HTML, the sequence `u, b` will lead to the following formatting: `<u><b>...</b></u>`.

These are all possible color logical names handled by source-highlight[3]:

```
black
red
darkred
brown
yellow
cyan
blue
pink
purple
orange
brightorange
green
brightgreen
darkgreen
teal
gray
darkblue
```

You can also use the direct color scheme for the specific output format, e.g., the `#<number>` syntax for specifying a color in HTML.

[2] Before version 2.1, this command line option was called `--tags-file` which used to be a very obscure name. I hope this name convention is a better one :-).

[3] You can see these colors in HTML in the file '`colors.html`'.

5.2 Language map

This configuration file associates a file extension to a specific language definition file. You can also use such file extension to specify the `--src-lang` option (see Chapter 4 [Simple Usage], page 7). Source-highlight comes with such a file, called '`lang.map`'.

Of course, you can override the settings of this file by writing your own language map file and specify such file with the command line option `--lang-map`). Moreover, as explained above, if a file '`lang.map`' is present in the current directory, such version will be used. The format of such file is quite simple:

```
extension = language definition file
```

The default language definition file is shown in Chapter 1 [Introduction], page 1.

5.3 Language definition files

These files are crucial for source-highlight since they specify the source elements that have to be highlighted. These files also allow to specify your own language definitions in order to deal with a language that is not handled by source-highlight[4]. The syntax for these files is explained in Chapter 7 [Language Definitions], page 14.

5.4 Output Language map

This configuration file associates an output format to a specific output language definition file. You can use the name of that output format to specify the `--out-format` option (see Chapter 4 [Simple Usage], page 7). Source-highlight comes with such a file, called '`outlang.map`'.

Of course, you can override the settings of this file by writing your own output language map file and specify such file with the command line option `--outlang-map`). Moreover, as explained above, if a file '`outlang.map`' is present in the current directory, such version will be used. The format of such file is quite simple:

```
output format name = language definition file
```

The default language definition file is shown in Chapter 1 [Introduction], page 1.

In particular, there is a convention for the output format name in the output language map, according to the suffix of the name with a dash `-`:

`-doc` The one used when `--doc` command line option is given

`-css-doc` The one used when `--css` command line option is given

`-css` The one used when `--css` and `--no-doc` command line options are given

If a combination of the above mentioned command line options is given for a specific output format, and a corresponding definition file is not specified in the map file, then an error is raised.

For instance, if you specified the definition file for your language `mylang` and also one for dealing with `--doc` option, i.e., a definition file for `mylang-doc`, and you run source-highlight as follows:

```
source-highlight -f mylang --css mycss.css
```

You will get the following error:

```
source-highlight: output language mylang-css-doc not handled
```

[4] This is the main difference introduced in version 2.0 with respect the the previous version.

5.5 Output Language definition files

These files are crucial for source-highlight since they specify how the source elements are highlighted. These files also allow to specify your own output format definitions in order to deal with an output format that is not handled by source-highlight[5]. The syntax for these files is explained in Chapter 8 [Output Language Definitions], page 21.

5.6 Developing your own definition files

I encourage those who write new language definitions or correct/modify existing language definitions to send them to me so that they can be added to the source-highlight distribution!

Since these files require more explanations (that, however, are not necessary to the standard usage of source-highlight), they are carefully explained in separate parts: Chapter 7 [Language Definitions], page 14 and Chapter 8 [Output Language Definitions], page 21.

[5] This is the main difference introduced in version 2.1 with respect the the previous version.

6 Invoking source-highlight

The format for running the source-highlight program is:

```
source-highlight option ...
```

source-highlight supports the following options, shown by the output of source-highlight --help:

```
source-highlight

Highlight the syntax of a source file (e.g. Java) into a specific format (e.g.
HTML)

Usage: source-highlight [OPTIONS]...

  -h, --help                    Print help and exit
  -V, --version                 Print version and exit
  -i, --input=filename          input file. default std input
  -o, --output=filename         output file. default std output. If STDOUT is
                                  specified, the output is directed to standard
                                  output
  -s, --src-lang=STRING         source language (use --lang-list to get the
                                  complete list).  If not specified, the source
                                  language will be guessed from the file
                                  extension.
      --lang-list               list all the supported language and associated
                                  language definition file
      --outlang-list            list all the supported output language and
                                  associated language definition file
  -f, --out-format=STRING       output format (use --outlang-list to get the
                                  complete list)  (default=‘html’)
  -v, --verbose                 verbose mode on
  -d, --doc                     create an output file that can be used as a
                                  stand alone document (e.g., not to be
                                  included in another one)
      --no-doc                  cancel the --doc option even if it is implied
                                  (e.g., when css is given)
  -c, --css=filename            the external style sheet filename.  Implies
                                  --doc
  -T, --title=STRING            give a title to the output document.  Implies
                                  --doc
  -t, --tab=INT                 specify tab length.  (default=‘8’)
  -H, --header=filename         file to insert as header
  -F, --footer=filename         file to insert as footer
      --style-file=filename     specify the file containing format options
                                  (default=‘default.style’)
  -n, --line-number             number all output lines
      --line-number-ref[=prefix]
                                number all output lines and generate an anchor,
                                  made of the specified prefix + the line
                                  number  (default=‘line’)
      --output-dir=path         output directory
      --gen-version             put source-highlight version in the generated
                                  file  (default=on)
      --lang-def=filename       language definition file
      --lang-map=filename       language map file  (default=‘lang.map’)
      --outlang-def=filename    output language definition file
      --outlang-map=filename    output language map file
                                  (default=‘outlang.map’)
      --data-dir=path           directory where language definition files and
                                  language maps are searched for.  If not
                                  specified these files are searched for in the
                                  current directory and in the data dir
                                  installation directory
      --check-lang=filename     only check the correctness of a language
```

```
                                definition file
--check-outlang=filename  only check the correctness of an output
                                language definition file
--failsafe                if no language definition is found for the
                                input, it is simply copied to the output
--debug-langdef           debug a language definition
```

Let us explain some options in details (apart from those that should be clear from the `--help` output itself, and those already explained in Chapter 4 [Simple Usage], page 7).

`--doc`
`-d`
: If you want a stand alone output document (i.e., an output file that is not thought to be included in another document), specify this option (otherwise you just get some text that you can paste into another document). If you choose this option and do not provide a `--title`, the your source file name will be used as the title.

`--no-doc`
: The `--doc` option above is actually implied by other command line options (e.g., `--css`). If you do not want this (e.g., you want to include the output in an existing document containing the global style sheet), you can disable this by using `--no-doc`.

`--css`
`-c`
: Specify the style sheet file (e.g., a '`.css`' for HTML[1]) for the output document.

`--tab`
`-t`
: With this options, tab characters will be converted into specified number of space characters (tabulation points will be preserved). This option is automatically selected when generating line numbers.

`--output-dir`
: You can pass to source-highlight more than one input file (see Chapter 4 [Simple Usage], page 7). In this case you cannot specify the output file name. In such cases the output files will be automatically generated into the directory where you invoked the command from; if you want the output files to be generated into a different directory you can use this option.

`--line-number-ref`
: As `--line-number`, this option numbers all the output lines, and, additionally, generates an anchor for each line. The anchor consists of the specified prefix (default is `line`) and the line number (e.g., `line25`). For instance, as prefix, if you deal with many files, you can use the file name. Notice that some output languages might not support this feature (e.g., `esc`, since it makes no sense in such case). See Section 8.4 [Anchors], page 23 for defining how to generate an anchor in a specific output language.

`--failsafe`
: If no language specification is found, an error will be printed and the program exits. With this option, instead, in such situations, the input is simply copied as it is to the output. This is useful when `source-highlight` is used with many input files, and it is also used in the '`src-hilite-lesspipe.sh`' script, Section 2.5 [Using source-highlight with less], page 5.

`--debug-lang`
: Allows to debug a language definition file, Section 7.10 [Debugging], page 19.

[1] As explained before, originally Source-highlight was thought mainly for generating HTML output, this is why the term *css* is used for style sheets.

7 Language Definitions

Since version 2.0 source-highlight uses a specific syntax to specify source language elements (e.g., keywords, strings, comments, etc.). Before version 2.0, language elements were scanned through Flex. This had the drawback of writing a new flex file to deal with a new language; even worse, a new language could not be added "dynamically": you had to recompile the whole source-highlight program.

Instead, now, language elements are specified in a file, loaded dynamically, through a (hopefully) simple syntax. Then, these definitions are used internally to create, on-the-fly, regular expressions that are used to highlight the elements. In particular, we use the regular expressions provided by the Boost library (see Chapter 2 [Installation], page 3). Thus, when writing a language definition file you will surely have to deal with regular expressions. Of course, we use the Boost regex library regular expression syntax. We refer to Boost documentation for such syntax, http://www.boost.org/libs/regex/doc/syntax.html.

Here, we see such syntax in details, by relying on many examples. This allows a user to easily modify an existing language definition and create a new one. These files have, typically, extension '`.lang`'.

Each definition basically associates a regular expression to a language element and defines a name for the language element. Such name will be used to associate a particular style (e.g., bold face, color, etc.) to the highlighting of such elements. You cannot use names that are the same of keywords used in the language definition syntax (e.g., `start`, as shown later, is a reserved word).

Comments can be given by using `#`; the rest of the line is considered as a comment.

7.1 Simple definitions

The simpler way of specify language elements is to list the possible alternatives. This is the case, for instance, for keywords. For instance, in '`java.lang`' you have:

```
keyword = "abstract|assert|break|case|catch|class|const",
          "continue|default|do|else|extends|false|final",
          "finally|for|goto|if|implements|instanceof|interface"
keyword = "native|new|null|private|protected|public|return",
          "static|strictfp|super|switch|synchronized|throw",
          "throws|true|this|transient|try|volatile|while"
```

The elements must be specified in double quotes. You can separate quoted definitions with commas. Alternatively, within a quoted definition, alternatives can be separated with the pipe symbol `|`. The above definition defines the language element `keyword`. Each time an element is found in the source file, it is highlighted with the style for the element with the same name in the output format style file (notice that all elements shown in the example are take from the language definition files that come with source-highlight and there is a style for each of such elements, see Chapter 5 [Configuration files], page 8). If such an element is not specified in the output format style file, it is simply not highlighted (so pay attention to typos :-).

From the above example you may have noticed that language element definitions are cumulative, so the second `keyword` definition does not replace the first one. (Indeed, in some case you may want to actually redefine a language element; this is possible as explained in the following sections.)

Notice that words specified in double quotes have to match exactly in a source file, and they must be isolated (not surrounded by anything but spaces). Thus for instance `class` is matched as a keyword, but in `my_class` the substring `class` is not matched as keyword. From the point of view of regular expressions a string such as `class` in a double quote simple definition is intended as `\<(class)\>`.

Special characters have to be escaped with the character \. So for instance if you want to specify the character |, which is normally used to separate alternatives in double quoted strings, you have to specify \|.

Definitions in double quotes are interpreted literally (thus, e.g., a dot . is interpreted as the character . not as the regular expression wild card). If you want to enjoy the full power of regular expressions to specify a language alternative, you have to use single quoted strings ('), instead of double quoted strings.

For instance, the following is the definition for a preprocessor directive in C/C++:

```
preproc = '^[[:blank:]]*#([[:blank:]]*[[:word:]]*)'
```

Notice that the definition 'class' is different from "class", as explained above. Thus, for instance 'class' matches also the sub-expression class inside my_class.

Finally, at the end of a list of definitions, one can specify the keyword nonsensitive; in that case, the specified strings will be interpreted in a non case sensitive way. For instance, we use this feature in Pascal language definition, 'pascal.lang' where keywords are parsed in a non sensitive way:

```
keyword = "alfa|and|array|begin|case|const|div",
      "do|downto|else|end|false|file|for|function|get|goto|if|in",
      "label|mod|new|not|of|or|pack|packed|page|program",
      "put|procedure|read|readln|record|repeat|reset|rewrite|set",
      "text|then|to|true|type|unpack|until|var|while|with|writeln|write"
  nonsensitive
```

7.2 Line wide definitions

It is often useful to define a language element that affects all the remaining characters up to the end of the line. For such definitions, instead of the = you must use the keyword start. For instance, the following is the definition of a single line comment in C++:

```
comment start "//"
```

This says that when the two characters // are encountered in the source file, everything from these characters, include, up to the end of the line, will be highlighted according to the style comment.

7.3 Order of definitions

It is important to observe that the order of language definitions is important since it will be used during regular expression matching. You then have to make sure that, if there are definitions that start with same characters, the longest expression is specified first in the file. For instance if you write

```
symbol = "/"
comment start "//"
```

The first expression will always be matched first, and the second expression will never be matched. The right order is

```
comment start "//"
symbol = "/"
```

7.4 Delimited definitions

Many elements are delimited by specific character sequences. For instance, strings and multiline comments. The syntax for such an element definition is

```
<name> delim <left delimited> <right delimiter> \
        {escape <escape character>} \
        {multiline} {nested}
```

The `escape` specification allows to specify the escape character that may precede one of the delimiters inside the element. This is optional.

For instance, this is the definition of C-like strings:

```
string delim "\"" "\"" escape "\\"
```

Notice that `\` is a special characters in definitions so it has to be escaped. If the `escape` specification was omitted, the C string `"write \"hello\" string"` would have been highlight incorrectly (it would have been highlighted as the string `"write \"`, the normal character sequence `hello\` and the string `" string"`).

The option `multiline` specifies that the element can spawn multiple lines. For instance, PHP strings are defined as follows:

```
string delim "\"" "\"" escape "\\" multiline
```

The option `nested` instructs to count possible multiple occurrences of delimited characters and to match relative multiple occurrences. For instance, C-like multiline comments are specified as follows:

```
comment delim "/*" "*/" multiline nested
```

If `nested` was not used the following nested comment would have not been highlighted correctly:

```
/*
   This is a /* nested comment */
*/
```

As said above, definitions are cumulative, and they are also cumulative even when using different syntactic forms. Thus, for instance, the complete definition for C++-style comments are the following:

```
comment start "//"
comment delim "/*" "*/" multiline nested
```

7.5 Variable definitions

It is possible to define variables to be re-used in many parts in a language definition file. A variable is defined by using

`vardef` <name of the variable> = <list of definitions>

Once defined, a variable can be used by prepending the symbol `$` to its name. For instance,

```
vardef FUNCTION = '(?:[[:alpha:]]|_)[[:word:]]*[[:blank:]]*(?=\()'
function = $FUNCTION
```

The capital letters are used only for readability.

It is also possible to concatenate variables and expressions, and reuse variables inside further variable definitions:

```
vardef basic_time = '[[:digit:]]{2}:[[:digit:]]{2}:[[:digit:]]{2}'
vardef time = '\<' + $basic_time + '\>'
```

7.6 File inclusion

It is possible to include other language definition files into another file. This is inclusion actually physically includes the contents of the included file into the current file during parsing, at the exact point of inclusion (just like the `#include` in C/C++). This is useful for re-using definitions in many files. For instance, C++ comment definitions are given in a file '`c_comment.lang`', and this file is included in the Java and C++ definition files. The same happens for number and functions. For instance, the file '`java.lang`' contains the following include instructions:

```
include "c_comment.lang"

include "number.lang"

keywords ...

include "function.lang"
```

Notice that the order of inclusion is crucial since the order of definition is crucial. If function definition was included before keyword definitions, then the sentence `if (exp)` would be highlighted as a function invocation.

7.7 State/Environment Definitions

Sometimes you want some source element to be highlighted only if they are surrounded by other elements. Source-highlight language definitions provides also this feature.

```
state|environment <standard definition> begin
  <other definitions>
end
```

This structure is recursive (so other state/environment definitions can be given within a state/environment). The meaning of a state/environment is that the definitions within the `begin ... end` are matched only if the definitions that define the state/environment have been matched. When entering a state/environment, however, the definitions given outside the state/environment are not matched. The difference between `state` and `environment` is that in the latter, normal parts of the source language (i.e., those that do not match any definition) are highlighted according to the style of the definition that defines the environment.

As an example, the following defines the multiline nested C comment, and highlights URL and e-mail addresses only when they appear inside a comment (notice that this uses file inclusion):

```
environment comment delim "/*" "*/" multiline nested begin
      include "url.lang"
end
```

Notice that we used `environment` because everything else inside a comment has to be formatted according to the comment style.

While for programming language definitions states/environments can be avoided, they are pretty important for highlighting files such as logs and ChangeLog files, since elements have to be highlighted when they appear in a specific position. For instance, for ChangeLog (see '`changelog.lang`'), we use a state for highlighting the date, name, e-mail:

```
state date start '[[:digit:]]{2,4}-?[[:digit:]]{2}-?[[:digit:]]{2}' begin
  string = '<(?:[[:word:]]*|\.)+@(?:[[:word:]]*|\.)+>'
  url = '(?:[[:word:]]|[[:punct:]])+'
end
```

Notice that definitions that appear inside a state/environment have the same scope of the expressions that define the environment. While this makes sense for `start` and `delim` definitions, it may makes less sense for simple definitions (i.e., those that simply lists all possible expressions):

in fact, in this case, such expressions do not define a scope. For such definitions, the semantics of state/environment is that the state/environment starts after matching one of the alternatives. And where will it end? In this case you must explicitly exit the environment. For instance, you can say that, when inside a state/environment, a specific language definition, when encountered also exits the environment (with the keyword `exit`). You can even exit all the environments with `exitall`. For instance, the following definition, highlights a non empty string following a web method:

```
vardef non_empty = '[^[:blank:]]+'

state webmethod = "OPTIONS|GET|HEAD|POST|PUT|DELETE",
          "TRACE|CONNECT|PROPFIND|MKCOL|COPY|MOVE|LOCK|UNLOCK" begin
  string = $non_empty exit
end
```

If you ever need such advanced features, you may want to take a look at the '`log.lang`' definition file that defines highlighting for several log files (access logs, Apache logs, etc.).

7.8 Redefinitions and Substitutions

These two features are useful when you want to define a language by re-using an existing language definition with some changes. Typically you `include` another language definition file and you redefine/substitute some elements.

When you use `redef` you erase all the previous definitions of that language elements with the new one. The new language element definition will be placed exactly in the point of the new definition. We use this feature, for instance, when we define the `sml` language by re-using the `caml` one: they differ only for the keywords[1]. In fact, the contents of '`sml.lang`' is summarized as follows:

```
include "caml.lang"

redef keyword = "abstraction|abstype|and|andalso..."

redef type = "int|byte|boolean|char|long|float|double|short|void"
```

Since the new language element definition appears in the exact point of the redefinition, this means that such a regular expression will be matched only if all the previous ones (the ones of the included file) cannot be matched. This may lead to unwanted results in some cases (not in the `sml` case though). In other words the following code

```
keyword = "foo"
keyword = "bar"
type = "int"
redef keyword = "myfoo"
```

is equivalent to the following one

```
type = "int"
keyword = "myfoo"
```

If this is not what you want, you can use `subst`, which is similar to `redef` apart from that it replaces the previous first definition of that language element in the exact point of that first definition (all other possible definitions are simply erased). That is to say that the following code

```
keyword = "foo"
keyword = "bar"
```

[1] At least, to the best of my knowledge :-)

```
type = "int"
subst keyword = "myfoo"
```

is equivalent to the following one

```
keyword = "myfoo"
type = "int"
```

It is up to you to decide which one fits best your needs. We use this feature to define `javascript` in terms of `java`:

```
include "java.lang"

subst keyword = "abstract|break|case|catch|class|if..."
```

Here using `redef` would have led to the unwanted behavior that `if (exp)` would have been highlighted as a function call, since the function element definition would have come first (and then matched first) than the redefinition of `if` as a keyword.

7.9 Concluding Remarks

By mixing all these features you can unleash your imagination and define highlighting for complex source languages such as Flex and Bison by writing few lines of code and re-use existing ones. For instance, Flex and Bison have their own syntax and lets you write C/C++ code in specific parts of the source language, e.g., the code between the outmost brackets, in the following example, is C++ code, and should be highlighted following C++ language definitions (apart from variables that are prefixed with `$`):

```
globaltags : options { if (...) { setTags( $1 ); } }
```

This is easy to do (taken from '`flex.lang`'):

```
state cbracket delim "{" "}" multiline nested begin
  variable = '\$.'
  include "cpp.lang"
end
```

Notice that, since we used `nested` we can be sure that the C++ language definitions are not considered anymore when we matched the last closing `}`.

7.10 Debugging

When writing a language definition file, it is quite useful to be able to debug it (by using complex regular expressions one may experience unwanted behaviors). Since version 2.1 the command line option `--debug-lang` is available. When using this option, some additional information are printed to the standard output.

When using this command line option the additional information produced has the following format:

```
<.lang filename>:<line number>: <matched subexpression>
formatting: <source file string to be formatted>
entering: <next state's regular expression>
exiting:
exitingall:
```

The lines starting with `entering`, `exiting` and `exitingall` are related to entering a new state/environment and exiting one and all states/environments. The first line shows a link to the '`.lang`' definition file and the line number, i.e., and the sub-expression that matched and the line starting with `formatting` shows the source file string that matched with that expression. If a line starting with `formatting` is not preceded by a line with the link to the sub-expression,

it means that no particular regular expression has matched, and thus the style `normal` will be used to format that string.

Consider the following (simplified) Java source file:

```
01: /*
02:   This is to demonstrate –debug-lang
03:  http://www.lorenzobettini.it
04: */
05:
06: package hello;
07:
08: public class Hello {
09:          // just some greetings ;-)  /*
10:     int i = 10;
11:     System.out.println("Hello World!");
12: }
```

Now you can debug the '`java.lang`' file by using the `--debug-lang` command line option. And the output is as follows:

```
c_comment.lang:11: (/\*)
formatting: /*
entering: (\*/)|(/\*)|...
formatting:
formatting:   This is to demonstrate --debug-lang
formatting:
url.lang:2: ((?:(?:[[:word:]]+://(?:[[:word:]]+[\./\-_]?)+)))
formatting: http://www.lorenzobettini.it
formatting:
c_comment.lang:11: (\*/)
formatting: */
exiting:
formatting:
formatting:
java.lang:1: (\<(?:import|package)\>)
formatting: package
formatting:  hello
symbols.lang:1: ((?:~|!|%|\^|\*|\(|\)|...
formatting: ;
... omitted ...
c_comment.lang:1: (//)
formatting: //
entering: (\z)
formatting:  just some greetings ;-)  /*
c_comment.lang:1: (\z)
formatting:
```

This should provide enough information to understand how the regular expressions are used and how the states/environments are entered and exited. Please notice that the sub-expressions that are shown may differ from the original ones specified in the '`.lang`' file. This is due to the preprocessing that is performed by Source-highlight. Moreover, some sub-expressions are not defined at all in the '`.lang`' file: for instance, this is the case for line wide definitions, i.e., those that are defined with the keyword `start`, Section 7.2 [Line wide definitions], page 15. The last lines above, showing `entering: (\z)`, mean that we wait to reach the end of a line.

8 Output Language Definitions

Since version 2.1 source-highlight uses a specific syntax to specify output formats (e.g., how to format in HTML, LaTeX, etc.). Before version 2.1, in order to add a new output format, many C++ classes had to be written. This had the drawback that a new output format could not be added "dynamically": you had to recompile the whole source-highlight program.

Instead, now, an output format is specified in a file, loaded dynamically, through a (hopefully) simple syntax. Then, these definitions are used internally to create, on-the-fly, text formatters.

Here, we see such syntax in details, by relying on many examples. This allows a user to easily modify an existing output format definition and create a new one. These files have, typically, extension '`.outlang`'.

Each definition basically associates a text style (such as, e.g., bold, italics, colors, etc.) to the representation of that style into the output format (such as, e.g., `<b>$text</b>` in HTML). The representation is given in `"` and you can use the classic escape character `\` to use the `"` inside the definition. If you want to specify the ASCII code for a character you can do so by specifying the numeric code in hexadecimal notation preceded by `\x`, for an example, see Section 8.6 [Style template], page 23.

If no definition is given for a specific style, e.g., bold, then when that style is requested during formatting, the text will be formatted as it is, i.e., the style without the definition is simply ignored.

Comments can be given by using `#`; the rest of the line is considered as a comment.

Files can be included in the same way as for language definitions, Section 7.6 [File inclusion], page 17.

In any case, if a definition for a style is given more than once, the last definition replaces all the others.

8.1 File extension

With the line:

```
extension "<file extension>"
```

you define the default file extension (without the `.`) used to generate files formatted according to this output format. This is used when no output file name is specified; if the file extension is not included in the `.outlang` is not defined, and no output file name is specified, an error will occur.

For instance, this is used in '`html_common.outlang`':

```
extension "html"
```

8.2 Text styles

These are the text styles that one can define:

```
bold
italics
underline
notfixed
fixed
```

These, of course, correspond to the ones used to specify the output format style, Section 5.1 [Output format style], page 8.

These definitions, for instance, are from the HTML format definition:

```
bold "<b>$text</b>"
italics "<i>$text</i>"
underline "<u>$text</u>"
```

Inside a definition you use the special variable `$text` to specify where the actual text to be formatted has to be inserted. For instance, the definition of `bold` above says that if you need to format the keyword `class` in bold in HTML, the following text will be generated: `<b>class</b>`. This variable is used also when mixing more than one styles recursively, in particular if you want to format in bold and italics (i.e, first bold and then italics, or, in other words, the sequence `i, b` is used in the the output format style file, see Section 5.1 [Output format style], page 8), then first the text `class` is substituted for `$text` into `<b>$text</b>` and then the text `<b>class</b>` will be substituted for `$text` into `<i>$text</i>`, thus obtaining `<i><b>class</b></i>`.

8.3 Colors

The definition for using colors during formatting requires the definition for the color style:

```
color "..."
```

For instance, for HTML we have:

```
color "<font color=\"$style\">$text</font>"
```

Apart from the variable `$text` that we already saw, we have also the variable `$style`, that will be replaced with the actual color.

Source-highlight recognizes a number of color constants, see Section 5.1 [Output format style], page 8.

You then must associate a color constant to the color definition in the output format, through the `colormap` definition:

```
colormap
"color constant" "color representation"
"color constant" "color representation"
...
default "default color representation"
end
```

The `default` row (notice the absence of ") defines the color to be used in case a color constant is used during formatting, but it is not defined in the output format.

For instance, for HTML we have:

```
colormap
"green" "#33CC00"
"red" "#FF0000"
"darkred" "#990000"
"blue" "#0000FF"
"brown" "#9A1900"
"pink" "#CC33CC"
"yellow" "#FFCC00"
"cyan" "#66FFFF"
"purple" "#993399"
"orange" "#FF6600"
"brightorange" "#FF9900"
"brightgreen" "#33FF33"
"darkgreen" "#009900"
"black" "#000000"
"teal" "#008080"
"gray" "#808080"
```

```
"darkblue" "#000080"
default "#000000"
end
```

If your output format does not handle colors you can simply avoid the definitions of `color` and `colormap` and Source-highlight will simply ignore colors.

The color is applied after applying the other styles, e.g., bold, italics, etc.

Thus, by continuing the example of the previous section, suppose you defined the following output style for keywords:

```
keyword blue i, b;
```

then the `class` text will be replaced to `$text` variable and the value `#0000FF` to `$style` inside the color definition `<font color="$style">$text</font>` obtaining `<font color="#0000FF">class</font>` which will then be replaced to `$text` in `<b>$text</b>` and so on for italics, finally obtaining

`<i><b><font color="#0000FF">class</font></b></i>`.

8.4 Anchors

When using the command line option `--line-number-ref` (Chapter 6 [Invoking source-highlight], page 12) an anchor is generated in the output file for each line numbering. The style of the anchor is defined by the definition `anchor`. If this is not defined, the option `--line-number-ref` has no effect. The `$style` variable will be replaced with the generated anchor, and the `$text` variable with the line number and a `:`.

For instance, for HTML we have

```
anchor "<a name=\"$style\">$text</a>"
```

8.5 One style

If the output format you are defining does not have a specific style for bold, italics, ... and for colors you can simply use the definition `onestyle`, where you can use both `$style` and `$text`. This will be used for any style (indeed any other definition such as bold, italics, color will be ignored). Indeed, in this case, it is assumed that the style of each source element is defined in a file with its own syntax, i.e., not with a syntax defined by Source-highlight. (This is the case, for instance, of HTML using CSS style sheets.) Moreover, since the output format style is not used, during formatting the variable `$style` will be replaced with the name of the element to highlight (e.g., `keyword`, `comment`, etc.).

For instance, for HTML CSS, we simply have:

```
onestyle "<span class=\"$style\">$text</span>"
```

In fact, HTML CSS relies on style definitions provided in a separate file (the '`.css`' file indeed). Thus, when formatting a `keyword`, e.g., `abstract`, we will obtain:

```
<span class="keyword">abstract</span>
```

Of course, the style for `keyword` must be defined in the '`.css`' file.

8.6 Style template

Some output formats are based on a unique template that where the other styles are composed; during composition the styles can be separated with a specific separator:

```
styletemplate "..."
styleseparator "..."
```

This is used, for instance, for the ANSI color escape sequence output format ('`esc.outlang`'):

```
styletemplate "\x1b[$stylem$text\x1b[m"
styleseparator ";"

bold "01$style"
underline "04$style"
italics "$style"
color "$style"
```

Notice that, since more than one style can be mixed into the style template, `bold`, `underline`, ... explicitly use the variable `$style`.

8.7 Character translation

Some characters that are in the source file may have a special meaning in an output format, so they need some preprocessing (e.g., escaping them). You can specify the translation table with:

```
translations
"original sequence" "transformed sequence"
"original sequence" "transformed sequence"
...
end
```

For instance, for HTML, we have the following translation table:

```
translations
"&" "&amp;"
"<" "&lt;"
">" "&gt;"
end
```

8.8 Document template

You can define the document template, i.e., the beginning and the end of an output file, with

```
doctemplate
"...beginning..."
"...end..."
end
```

For instance, for HTML we have

```
doctemplate
"<pre><tt>"
"</tt></pre>
"
end
```

Notice that in the end part there is an explicit new line.

In the definition of the `doctemplate` the following variables can be used and will be replaced during the output generation:

`$title` the value of the title for the output file (e.g., the one passed with the `--title` command line option;

`$header` the contents of the file specified with the command line option `--header`;

`$footer` the contents of the file specified with the command line option `--footer`;

`$css` the value passed with the command line option `--css`;

`$additional`
: other additional information. Source-highlight replaces this with its name and its version.

For instance, for an HTML document with css, (file '`cssdoc.outlang`') we have:

```
doctemplate
"<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0//EN\"
    \"http://www.w3.org/TR/REC-html40/strict.dtd\">
<html>
<head>
<meta http-equiv=\"Content-Type\"
content=\"text/html; charset=iso-8859-1\">
<meta name=\"GENERATOR\" content=\"$additional\">
<title>$title</title>
<link rel=\"stylesheet\" href=\"$css\" type=\"text/css\">
</head>
<body>
$header<pre><tt>"
"</tt></pre>
$footer</body>
</html>
"
end
```

9 Examples

Here we provide some examples of sources formatted with Source-highlight using the `-f texinfo` command line option. Please keep in mind that the highlighting will not be visible in the Info file, but only in the printed manual and in the HTML output (well, at least line numbers are visible everywhere :-).

The first example is produced by using the command:

```
source-highlight -f texinfo -i test.java -o test.java.texinfo -n
```

and here's the result

```
01: /*
02:   This is a classical Hello program
03:   to test source-highlight with Java programs.
04:
05:   to have an html translation type
06:
07:       source-highlight -s java -f html –input Hello.java –output Hello.html
08:       source-highlight -s java -f html < Hello.java > Hello.html
09:
10:   or type source-highlight –help for the list of options
11:
12:   written by
13:   Lorenzo Bettini
14:   http://www.lorenzobettini.it
15:   http://www.gnu.org/software/src-highlite
16: */
17:
18: package hello;
19:
20: import java.io.* ;
21:
22: /**
23:  * <p>
24:  * A simple Hello World class, used to demonstrate some
25:  * features of Java source highlighting.
26:  * </p>
27:  *
28:  * @author Lorenzo Bettini
29:  * @version 2.0
30:  */
31: public class Hello {
32:     int foo = 1998 ;
33:     int hex_foo = 0xCAFEBABE;
34:     boolean b = false;
35:     Integer i = null ;
36:     char c = '\'', d = 'n', e = '\\' ;
37:     String xml = "<tag attr=\"value\">&auml;</tag>", foo2 = "\\" ;
38:
39:     public static void main( String args[] ) {
40:         // just some greetings ;-)  /*
41:         System.out.println( "Hello from java2html :-)" ) ;
42:         System.out.println( "\tby Lorenzo Bettini" ) ;
```

```
43:         System.out.println( "\thttp://www.lorenzobettini.it" ) ;
44:         if (argc > 0)
45:             String param = argc[0];
46:         //System.out.println( "bye bye... :-D" ) ; // see you soon
47:     }
48: }
```

10 Reporting Bugs

If you find a bug in `source-highlight`, please send electronic mail to

`bug-source-highlight at gnu dot org`

Include the version number, which you can find by running '`source-highlight --version`'. Also include in your message the output that the program produced and the output you expected.

If you have other questions, comments or suggestions about `source-highlight`, contact the author via electronic mail (find the address at http://www.lorenzobettini.it). The author will try to help you out, although he may not have time to fix your problems.

11 Mailing Lists

The following mailing lists are available:

`help-source-highlight at gnu dot org`

for generic discussions about the program and for asking for help about it (open mailing list), http://mail.gnu.org/mailman/listinfo/help-source-highlight

`info-source-highlight at gnu dot org`

for receiving information about new releases and features (read-only mailing list), http://mail.gnu.org/mailman/listinfo/info-source-highlight.

If you want to subscribe to a mailing list just go to the URL and follow the instructions, or send me an e-mail and I'll subscribe you.

Concept Index

(Index is nonexistent)