Tutorial: Adding Language Support to SlickEdit

May 14, 2008

by Dennis B

In this article, I am going to illustrate how to add very simple support for a language that, out of the box, is not already handled by SlickEdit. To this end, I am choosing "Logo" as an example. I made this choice partially as a follow-up to Scott\'s article about using it as an educational language. At first, I was going to use LOLCODE as my example, but then a genuine fear fell over me that I would be contributing to the delinquency of programmers by putting together support for LOLCODE\'ing in SlickEdit.

I have used the freely available book "The Great Logo Adventure!" as a reference in this article. Please note that I am not an experienced Logo programmer, so if you bear with me, you can learn how to add language support and I can learn something about Logo.

Step 1: Defining the language

Go to Tools > Options..., and navigate to Languages > "Language Manager", then click on "Add Language..." Enter the following:

- Mode name: Logo
- Associated extensions: logo lgo
- Color coding: leave blank

Step 2: Color Coding

Now navigate to your Logo language settings and find the "Color coding" page.

Click on the top-most "New..." button to create a new Lexer and name it Logo. Now, we need to define the color coding. Let\'s work left to right, starting with the Tokens tab.

Logo is case-insensitive, so uncheck "Case sensitive". The identifier characters can be left pretty much as they are. Logo, in fact, will allow nearly anything to be an identifier, but in practical terms, the typical C language style identifiers are adequate.

Next, we add Keywords. The list below is fairly complete, though there is a fair amount of room for interpretation with respect to what is a keyword and what isn\'t in Logo. Note that when you add new Keywords, you can add several at once if you separate them with spaces.

```
AND CS IFTRUE OR SENTENCE ASCII CT LABEL PARSE SETUP ASK EDIT LEFT PR SHOW

BACK END LIST PRINT TC

BACKSLASHEDP EQUALP LOAD READLIST TEST

BYE FALSE LOCAL READWORD TO

CHAR FLASHER LOCALMAKE REPEAT TRUE

CLEARSCREEN FORWARD MAKE RUNPARSE TYPE CLEARTEXT IFELSE NOT RIGHT WAIT COUNT IFFALSE NUMBERP SAVE
WORD
```

Now switch to Preprocessing and add:

`.DEFMACRO .SETBF .EQ .SETFIRST .MACRO .SETITEM .MAYBEOUTPUT`

And Punctuation: `( ) : [ ]`

And Operators: `+ - / * =`

Now switch to Library Symbols.

This is where you would typically add coloring for functions that are built into the language or included with the standard language libraries. Since Logo has nearly 500 built-ins, it would be senseless to try to type them all in manually. The attached hotfix includes a file, "logo_builtins.txt" which contains all the built-ins. This file was constructed by opening the help file for MSWLogo (Microsoft Windows Logo, by Softtronics, Inc.) using SlickEdit\'s Help Indexer ("Help" > "Configure F1 Index Help"). I added the help file, then clicked on "Export..." to extract the index entries, then edited out the few things which were not library functions. In the Color Coding dialog, click on "Get...", point it at this file, and it will pull in all the words in the list file.

Note that the color coding engine is not perfect is the keyword coloring. Logo is very context dependent whether a symbol is being used as keyword or just a word or list. Lists are sometimes code, sometimes strings. The color coding engine has no idea about these rules, so it sees a word that matches a keyword and it colors it. This is simultaneously quite useful, quite annoying, and quite unavoidable.

We can now turn our attention to the Numbers tab. The default settings will be adequate here.

In the Strings tab, we have to do something ugly. Check the two items:

- Double quoted strings are always 1 char long
- Single quoted strings are always 1 char long

This is necessary because Logo doesn\'t have the typical kind of string constants we have grown to expect in other languages, so we need to trick it, otherwise nearly everything will look like strings. This solution is not ideal, but it is readable. Sometimes, unusual language features will not map to SlickEdit\'s default color coding engine, and you have to settle for the best you can do and move on. This is one such case.

The default settings are fine for the Language tab, so let\'s move on to Comments. Logo has one type of comment, the semicolon. So, click on "New Line Comment..." and specify that it uses ";" as the delimiter.

We are now done with configuring color coding. Color coding definition are saved in "user.vlx" in your configuration directory. I recommend opening "user.vlx" and copying the entire definition for [Logo], then saving it to a new file "logo.vlx", also in your configuration directory. This will be used later when we package up the language support.

Step 3: Tuning language settings

We can now configure other Language specific options for Logo. Start with the *Indent* options. You will probably want to set your tabs to "+3", or something more in line with your preferences.

Now switch to the *Comments* options. Define the "Comment line" to use "; " on the left so that you will be able to use "Document" > "Comment Lines" and "Document" > "Uncomment Lines" in your Logo source files.

On the *General* tab, make sure "Logo" is selected as the Color coding Lexer name. I also recommend turning on the "Current Line" check box to highlight the current line, since SlickEdit does current line and selection highlighting better than any editor out there.

While we are on the *General* tab, click on "Language-Specific Project..." to set up a command for loading and running Logo programs in MSWLogo. Go to the "Tools" command and select "Execute". Enter "logo32.exe -l

%f" for the Command Line, and specify the Run from dir to be "%p" (file path). You may have to give a [quoted] absolute path to logo32.exe if it is not on your regular path.

Step 4: Configure aliases

Switch to the *Aliases* options for Logo. This will allow you to define some typing shortcuts for Logo syntax. I will provide one example and let you go from there.

To define an alias for defining a function, click on "New..." and name the alias "to". Fill in the alias text with the following: `to %\\c %\\i%\\c end`

"%\\c" specifies a cursor landing and "%\\i" specifies indentation.

Step 5: Create a language support module

All of the above work displays how you can add support for viewing color coded files in SlickEdit, entirely through the configuration dialogs without writing any Slick-C code. However, if you want to hand what you have done over to another developer, the best way is to write a language support module that he can simply load and use. For what we have done so far with Logo, the essentials can be accomplished with the following small amount of code. A complete copy of the language support module (logo.e) is attached to this article.

```
#define LOGO_MODE_NAME "Logo" #define LOGO_EXTENSION1 "logo" #define LOGO_EXTENSION2 "lgo"

// This function is called when the module is loaded. // It configures the "logo" and "lgo" file
extensions. void defload() { _str setup_info="MN="LOGO_MODE_NAME",TABS=+3,MA=1 74 1,":+ "KEYTAB=ext-
keys,WW=1,IWT=0,ST=0,IN=2,":+ "WC=A-Za-z0-9_$,LN="LOGO_MODE_NAME",CF=1"; _str compile_info="0 logo32
*"; _str syntax_info="3 1 2 1 0 1 0"; _str be_info="(to)|(end);i"; int kt_index=0;
_CreateLanguage(kt_index, LOGO_MODE_NAME, setup_info, compile_info, syntax_info, be_info);
_CreateExtension("logo", LOGO_MODE_NAME); _CreateExtension("lgo", LOGO_MODE_NAME);

_str logo_vlx = _config_path() :+ FILESEP :+ "logo.vlx"; if (file_exists(logo_vlx)) {
import_lexer_file(logo_vlx); } }
```

Step 6: Write tagging support for Logo

Tagging is not rocket science, you can write your own tagging support with a minimal amount of effort, depending on the language you are trying to parse.

In the case of Logo, the critical item is the "to" statement, which defines a procedure.

User-defined tagging support is accomplished by writing a language-specific "proc-search" function, the signature for this function is as follows:

```
int logo_proc_search(_str &proc_name, int find_first);
```

The function returns <0 on error, and 0 on success, also setting "proc_name" to the encoded information about the symbol which was found.

The basic outline for a "proc-search" function is to start by searching for a regular expression which will find lines that symbol declarations happen on. If the search fails, return the failure status and it\'s all over. Otherwise, you parse apart the line and get the name of the symbol that was declared. Finally, you set proc_name and return 0. A very simple example for Logo is included below. The "logo_proc_search" included in logo.e is a little more thorough than this, but this is a good starting point for nearly any language.

```
int status = search("^(:b|)(to:b|make:b[\\"]):v","@rih>Xcs"); if (status) { proc_name=""; return
status; } get_line(line); parse strip(line) with kw proc_name args; switch (lowcase(kw)) { case "to":
proc_name = tag_tree_compose_tag(proc_name,"","proc",0,args); break; case "make": proc_name =
substr(proc_name,2) :+ "(gvar)"; break; } return(0);
```

If you want to support tagging local variables as well

as globals, you will need to implement the "list-locals" callback. It\'s signature is shown below. The implementation is quite similar to the "logo-proc-search" above, except that rather than working in a start and stop pattern, the list-locals function is just expected to find everything and insert the symbols directly using the tagging API function tag_insert_local2(). I will not detail it\'s implementation in this article -- please take a look at logo.e to learn more.

```
void logo_list_locals(int unused_output_view_id, _str unused_filename_p, _str embedded_ext, int
ltf_flags, int unused_tree_wid, int unused_bitmap_index, int cur_start_seekpos, int end_seekpos);
```

Step 7: Adding support for block matching

For many languages, the default block matching which supports parentheses, braces, brackets, and configurable begin-end pairs is adaquate. For other languages, you may want to do more. The prototype hook function for implementing advanced block matching is below. For an excellent example of how to implement this function, look at ada.e.

```
int _[lang]_find_matching_word(boolean quiet);
```

Step 8: Adding support for syntax expansion

The typical way to add support for syntax expansion is to write commands to handle [space] (syntax expansion) and [enter] (syntax indent). A simple example of this can be found in modula.e. By using the "ext_keys" event table, your language-specific space and enter handlers will be automatically hooked in. For this sample, I am not going to write syntax expansion for Logo -- I leave that as an exercise for the reader.

Step 9: Adding support for Context Tagging®

There are six major hook functions necessary to implement support for Context Tagging for a language. Again, I am not going to implement all of these for Logo, but leave that as an exercise for the reader. There are numerous examples in the Slick-CÃ‚Â® code of how to implement these functions for a variety of languages. Look at csymbols.e for examples of how they are implemented for a conventional language such as C++. Look at cobol.e for an example of how they can be implemented for less structured languages where you can have function calls without parens, for example.

`int _[lang]_MaybeBuildTagFile(int &tfindex);` This function is used to automatically build a tag file for language-specific libraries and built-in functions. You can create a [lang].tagdoc file to document built-in functions. Look at [slickedit]/builtins/basic.tagdoc or html.tagdoc for some examples of how this is done. The attached Logo sample attempts to build a tag file for Microsoft Windows Logo. Note that there is a small problem, because most of the files in their run-time library are extensionless, thus, not recognized immediately as Logo.

```
int _[lang]_get_expression_info(boolean PossibleOperator, VS_TAG_IDEXP_INFO &idexp_info);
```

This function is used to get information about the code at the current buffer location, including the current identifier under the cursor, the expression before the current identifier, and other supplementary information useful to Context Tagging. If the _[lang]_get_expression_info hook function is not implemented, the editor will use a default implementation which simply returns the current identifier under the cursor and no prefix expression.

```
int _[lang]_find_context_tags(_str (&errorArgs)[],_str prefixexp, _str lastid,int lastidstart_offset,
int info_flags,typeless otherinfo, boolean find_parents,int max_matches, boolean exact_match,boolean
case_sensitive, int filter_flags=VS_TAGFILTER_ANYTHING, int context_flags=VS_TAGCONTEXT_ALLOW_locals,
VS_TAG_RETURN_TYPE (&visited):[]=null, int depth=0);
```

Find tags matching the identifier at the current cursor position using the information extracted by {@link _[lang]_get_expression_info()}. This callback is used for symbol navigation, symbol completion, references, and symbol analysis.

```
int _[lang]_insert_context_tags(_str (&errorArgs)[], int editorctl_wid,_str prefixexp, _str
lastid,_str lastid_prefix,int lastidstart_offset, _str expected_type,int info_flags,typeless
otherinfo);
```

Insert tags into the list members tree control matching the given identifier and prefix expression.

If the _[lang]_insert_context_tags() or _[lang]_find_context_tags() hook functions are not implemented, the editor will use a default implementation which simply tries to find a symbol matching the current identifier under the cursor, ignoring any prefix expression.

```
int _[lang]_fcthelp_get_start( _str (&errorArgs)[], boolean OperatorTyped, boolean
cursorInsideArgumentList, int &FunctionNameOffset, int &ArgumentStartOffset, int &flags );

int _[lang]_fcthelp_get( _str (&errorArgs)[], VSAUTOCODE_ARG_INFO (&FunctionHelp_list)[], boolean
&FunctionHelp_list_changed, int &FunctionHelp_cursor_x, _str &FunctionHelp_HelpWord, int
FunctionNameStartOffset, int flags );
```

The _[lang]_fcthelp_get_start() callback is used to find the start of a function call. This determines quickly whether or not we are in the context of a function call. The _[lang]_fcthelp_get() callback then does all the heavy lifting to find the matching functions and parse out their parameter lists and set up all the information in order for the function parameter info feature to display a prototype for the current function.

If the _[lang]_fcthelp_get_start() and _[lang]_fcthelp_get() hook functions are not implemented, the editor will use a default implementation which simply assumes that function parameters are comma-separated lists surrounded by parenthesis.

Step 10: Creating a redistributable hot fix

An easy way to redistribute a set of macros and support files is to package them together using SlickEdit\'s hot fix mechanism. To package a hot fix for the Logo support all you would have to do is use the create-hotfix command from the SlickEdit command line. Note that this command requires you to have the Cygwin "zip" utility on your path. You can also construct a hot fix manually by creating your own hotfix.xml manifest and dragging files into a zip file.

```
create-hotfix [config]/logo.vlx [slickedit]/macros/logo.e
```

The Logo support can be loaded by going to Help > Product Updates > Load Hot Fix... and pointing it to the addons_se13000_logo.zip file.

Summary

This article gives you an initial glance at how to add support for a new language to SlickEdit. It does not detail every single aspect of adding language support. There is still a great deal for you to learn by studying some of the language support code shipped with SlickEdit, such as basic.e, awk.e, ansic.e, pascal.e, modula.e, and fortran.e.

Zip file can be found here