

Developing custom plug-ins for the Vim editor

Arpan Sen

November 09, 2010

Learn how to extend the popular and versatile Vim editor to suit your systems administration needs using Vim's custom scripting language and options such as Perl and Python.

Introduction

Vim is one of the two most popular editors in just about any flavor of UNIX®, despite its minimalist interface. You can easily extend it to suit a wide variety of software development and systems administration needs. Vim even has its own scripting language that you can use to code your scripts and then load them into Vim. Alternatively, you can use external scripting languages like Perl or Python to extend the editor's functionality. Collectively, these scripts are *Vim plug-ins*.

Syntax highlighting of programming languages is the most common situation where a custom plug-in would be helpful. Vim installation comes with a host of predefined syntax support for `c`, `c++`, Perl, and Tcl (look in *Vim_Installation_Folder/vim72/syntax*), but sometimes you need additional support for custom or new programming languages, or to extend the plug-ins to apply organization-specific coding standards.

Similarly, compiling sources from within the editor is a nice feature to have. Creating a custom plug-in for Perl or Python code that lets you compile sources from within the editor, and then positions the cursor on the error may help save a lot of development time.

This article shows what Vim has to offer to highlight syntax from a custom programming language. It enforces coding conventions by following simple regular expression usage and moves on to Perl scripting with Vim. The article ends by showing how to compile sources from inside Vim.

Note: This article assumes that you have basic familiarity with Vim, Perl, `make`, and regular expressions. You'll be using Vim version 7.2 and Perl version 5.8.

Syntax highlighting

We will use Vim's internal scripting engine to highlight syntax from a custom language you have created. [Listing 1](#) contains some of the keywords from this custom language.

Listing 1. Keywords from your custom programming language

```
foreach if then else elseif while repeat until disable
integer unsigned signed byte
always initial
```

Vim associates some word to be a keyword using the format: `syntax keyword <group name> <keyword list>`

So, for your custom language, use the pseudo-code in [Listing 2](#).

Listing 2. Defining keywords in Vim

```
syntax keyword group1 foreach if then else elseif while
repeat until disable integer unsigned signed byte always initial
```

Next, save this file as *lang.vim* under \$HOME. Now, edit a small snippet of code in your custom language (see [Listing 3](#)).

Listing 3. Code in the custom programming language

```
integer k=0;
repeat (k < 3) begin
    print "hello world" + k + "\n";
    k = k + 1;
end
```

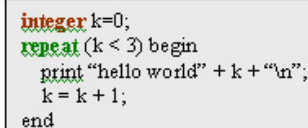
In the Vim editor, load lang.vim as `:source $HOME/lang.vim` and you are done. But wait, there is a problem, nothing happens. Even though you specified the syntax you did not provide inputs as to how it should be highlighted. [Listing 4](#) shows an improved version of the lang.vim file.

Listing 4. Improved version of lang.vim that supports syntax highlighting

```
syntax keyword type1 integer unsigned signed byte
syntax keyword statement1 foreach if then else elseif
while repeat until disable always initial
highlight link type1 Type
highlight link statement1 Statement
```

Reload lang.vim, and the code from [Listing 3](#) is now syntax highlighted (see [Figure 1](#)). In [Figure 1](#) below, the keywords *integer* and *repeat* are highlighted.

Figure 1. Highlighted syntax in the custom code



```
integer k=0;
repeat (k < 3) begin
    print "hello world" + k + "\n";
    k = k + 1;
end
```

What exactly did you do in [Listing 4](#)? There are two things to understand:

- The user would typically expect the statements in the program (`if-then-else`, `repeat`, and so on) highlighted differently from the data types (`integer`, `byte`) for better readability. So, you split the syntax into groups with appropriate contents: The group `type1` contains the `integer`, `unsigned`, `signed`, and `byte` keywords.

- Vim has predefined syntax groups like `Type`, `Statement`, `Comment`, and `Identifier` that come with their specific color schemes. The `highlight` command associates `type1` with Vim's `Type` group so that the same color scheme reflects for keywords like `byte`.

More syntax support

You would probably like your language to be case-neutral so that `integer` and `INTEGER` both get highlighted. You'd also like to add support for `//` C++-style comments. [Listing 5](#) shows the modified `lang.vim` file.

Listing 5. Improved version of `lang.vim` that supports syntax highlighting

```
syntax case ignore
syntax keyword type1 integer unsigned signed byte
syntax keyword statement1 foreach if then else elsif
    while repeat until disable always initial
syntax match comment1 /\s*\/.*/
highlight link type1 Type
highlight link statement1 Statement
highlight link comment1 Comment
```

The `syntax case ignore` statement takes care of the case neutrality. You cannot handle comments using keywords, so you need a regular expression that you can then associate with the Vim `comment` group. You define the regular expression using `syntax match <identifier> /<pattern>/`. In between the start and end forward slash (`/`), you defined the pattern `\s*\/.*`, which signified anything that started with `//` continued to the end of the line. So, the code shown in [Figure 2](#) from your custom language has the proper highlighting.

Figure 2. Support for comments and case independence

```
Integer k = 4; // this works
// so does this!
```

Custom coding standard support

You can easily extend the custom plug-in to handle organization-specific coding standards. Here are some typical guidelines:

- No tabs in the code.
- No variable name should be longer than 14 characters.
- Do not have more than 80 characters on a single line.
- Functions may not be longer than 100 lines.

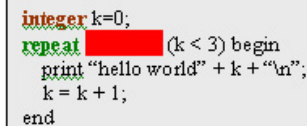
No tabs in the code

Let's begin with the simplest guidelines, no tabs in the code. You simply define an identifier for `tab` then associate that identifier with Vim's predefined `Error` tag:

```
syntax match identifier1 "\t"
highlight link identifier1 Error
```

So, what happens if you do have tabs in your code? [Figure 3](#) shows a revised version of the code in [Figure 2](#) with tabs.

Figure 3. Vim highlighting the existence of tabs in red



```
integer k=0;
repeat (k < 3) begin
  print "hello world" + k + "\n";
  k = k + 1;
end
```

In [Figure 3](#), the tab just after *repeat* is highlighted in red; a clear flag to the user that something is wrong.

No variable name longer than 14 characters

Support for variable name lengths requires more understanding of the use of regular expressions in syntax match:

```
syntax match longword1 "\w\{14,}"
highlight link longword1 Error
```

Here, `\w` defines the character class `[0-9A-Za-z_]`—that is, any digit, alphabetic character (uppercase or lowercase), or underscore (`_`) is allowed. The next sequence is `\{14,}`, which means that at least 14 consecutive occurrences need to be matched. So, `this_is_a_REAL_long_word1` will result in a highlight, as the identifier length is greater than 14, while `this_is_ok_2` is going to do just fine. [Figure 4](#) shows how an error situation looks.

Figure 4. Variable names longer than 14 characters are flagged in red



```
integer this_is_a_REAL_long_word1;
integer this_is_ok_2;
```

In [Figure 4](#), the variable `this_is_a_REAL_long_word1` is highlighted in red (again as per Vim's default color profile) warning the user that something is wrong.

No more than 80 characters on a single line

Having more than 80 characters on a single line adds to the clutter and makes reading difficult. You would again be using `syntax match` to define an identifier for this regular expression and link to `Error`. Figuring out the regular expression should not be difficult: The caret (`^`) signifies the start of a line; the dollar sign (`$`) signifies the end of a line; and anything in between should be longer than 80 characters for an error match. Note that the period (`.`) signifies a match for any character other than end-of-line:

```
syntax match longline1 "^.\{80,$}"
highlight link longline1 Error
```

[Figure 5](#) shows a code line that exceeds 80 characters in length and is therefore highlighted.

Figure 5. Rules to prevent line length from exceeding 80 characters

```
repeat(k < 3) begin
  k = k + 1; // this is a very interesting exercise
  mydata.mem[k] = mydata.mem[f(2*k+3) - f(2*k)] + sqrt(f(3.221 * k)) - 662.34;
end
```

Since we exceeded 80 characters thanks to the complex formula, in Figure 5, Vim highlights the whole line in red. The highlight will go the moment we enter a backspace and the line length reduces to 79 from 80.

No functions longer than 100 lines

The last in the list of custom coding conventions is that function length must always be less than 100 lines. [Listing 6](#) shows a function defined in your custom programming language.

Listing 6. A function defined in the custom programming language

```
function f (int k, int l) returns float
begin
  f = k * l;
  for (int i=0; i<10; i++)
  begin
    f += sqrt(k) * sqrt(l);
  end
  return f + 2;
endfunction
```

It's probably easier (and definitely faster) to have the entire function passed to Perl for checking the number of lines than to devise a complex regular expression or call Vim predefined internal functions. The next section has the details.

Using external scripting languages to create a Vim plug-in

Vim is easy to connect with Perl, Python, Tcl, and Ruby scripts. This discussion, however, is restricted to Perl, but interfacing with Python, Tcl, and Ruby is similar. [Listing 7](#) describes a Vim plug-in that displays an error message if the length of any function is greater than 100 lines.

Listing 7. Creating a custom plug-in for Vim using Perl

```
perl << EOF
sub checksize
{
  my $count = 0;
  my $startfunc = 0;
  my $filelen = scalar @_;
  while ($count < $filelen)
  {
    if ($_[ $count ] =~ /^function/)
    {
      $startfunc = $count;
    }
    elsif ($_[ $count ] =~ /endfunction/)
    {
      if ($count - $startfunc > 100)
      {
        Vim::Msg($_[ $startfunc ], "Error");
      }
    }
    $count++;
  }
}
```

```

    }
    }
    ++$count;
  }
}
EOF

function! L1( )
  perl checksize($curbuf->Get(1..$curbuf->Count()))
endfunction

```

All of this is coded in the same lang.vim file you used earlier. Here are the nuances of the plug-in:

1. You embed Perl code inside Vim script using markers. These markers could have any name and may not be in all uppercase characters. In [Listing 7](#)'s `perl << EOF ... EOF`, `EOF` was the marker used. Make sure that the second `EOF` begins at the first column of a line. The marker does not have to be named `EOF`—any name would do—but the first-column rule is sacrosanct.
2. The entire contents of the file are passed to the Perl code. The Perl subroutine `checksize` traverses over the whole file (as part of the `@_ implicit` array in Perl) and keeps checking for function lengths. When it encounters the keyword `function`, it sets a counter to 0; when it hits on the `endfunction` keyword, it checks whether the counter is greater than 100. If the function length exceeds 100, an error message is displayed.
3. You cannot use the usual Perl print routines to display the error message, because you need the message to be displayed inside Vim. Vim provides a useful interface to Perl, the details of which are available in [Resources](#). `Vim::Msg` displays the message inside the editor window. In [Listing 7](#), you display the first line of the offending function. The second argument to `Vim::Msg` is the type of information that is displayed: `error` implies that this information is highlighted in red.
4. You define a function in Vim that passes to the Perl code the file sources. `$curbuf->Count()` tells you how many lines there are in the current buffer; `$curbuf->Get(<line1>..<line2>)` returns the text between the lines specified by `line1` and `line2`. In the script, you are passing the contents from line 1 to last line of the current buffer. Now, in ESC mode, type `:call L1()`: You should immediately see the offending function(s) listed.

Compiling sources from within Vim

Vim makes it possible to compile sources from inside the editor. Together with syntax highlight and custom code checks, this feature makes Vim as close to a custom integrated development environment (IDE) as it gets. [Listing 8](#) has a `c++` file that contains a few errors.

Listing 8. Really messed up C++ code

```

#include <iostream>
using namespace stdl

class mytags {
public:
  int getid(int id=0);
  void setid(int)
protected:
  list<int> tags;
  const list<int>::iterator tag_i;
}

```

[Listing 9](#) provides a cool five lines to add to your Vim script for compiling inside the editor.

Listing 9. Mapping the F3 key to compile and display errors inside editor

```
function! build()
  make
  cl "list the errors
endfunction
map <F3> :call build(<CR>
```

Press the F3 key in ESC mode to compile the sources. The `build()` function calls `make` from inside Vim, and then invokes `cl`, which displays the errors. To go to the first error, type `:cfirst` in ESC mode; to go to every subsequent error, use `:cn`; to go to the last error, type `:clast`. Note that by default, the `Makefile` is assumed to be in the same folder as the sources. That said, none of this is necessary, because you can modify the `build()` function from [Listing 9](#) to go to the folder in which the `Makefile` resides. Also, it is easy enough to pass arguments to `make`. [Listing 10](#) clarifies the point.

Listing 10. An improved Vim script for building sources using make

```
function! build()
  cd /home/arpan/ibm/scripts "go to the folder where Makefile is
  make CC=g++
  cd /home/sources "back to sources
  cl "list the errors
endfunction
map <F3> :call build(<CR>
```

[Listing 11](#) has the errors that now show up inside Vim.

Listing 11. Build errors inside the Vim environment

```
#include <iostream>
using namespace stdl

class mytags {
public:
  int getid(int id=0);
  void setid(int)
protected:
  list<int> tags;
  const list<int>::iterator tag_i;
}

t.cpp:4: error: expected namespace-name before "class"
t.cpp:4: error: `<type error>' is not a namespace
t.cpp:4: error: expected `;' before "class"
t.cpp:8: error: expected `;' before "protected"
t.cpp:10: error: ISO C++ forbids declaration of `list' with no type
t.cpp:10: error: expected `;' before '<' token
t.cpp:6: error: expected unqualified-id at end of input
t.cpp:6: error: expected `,' or `;' at end of input
Press ENTER or type command to continue
```

Installing a syntax file

Create a folder named *syntax* under `$HOME/.vim`, and copy the custom plug-in to it. If your custom language is called *m12*, say, then name this file *m12.vim*. Then, edit `$HOME/.vimrc` and add the

line `syntax on`. That's it: Now, whenever you open a file with the extension `m/2` in vim, the syntax is automatically highlighted. This behavior does not include explicit function calls; it's advisable to have key mappings for quick custom code checks, such as subroutine length.

Conclusion

A high-flying integrated IDE is not a necessity for rapid action development, and Vim amply testifies for this philosophy. With interfaces to languages like Ruby and Python, it's even possible to connect to the web with Vim. Refer to the [Resources](#) section for further reading on the subject.

© Copyright IBM Corporation 2010

(www.ibm.com/legal/copytrade.shtml)

[Trademarks](#)

(www.ibm.com/developerworks/ibm/trademarks/)