Lazarus to the aid of Visual Studio Code

Michaël Van Canneyt

May 29, 2023

Abstract

Lazarus has excellent code tools. VS Code has a framework for adding support for new languages. In this article we show how first-class pascal support can be implemented in Visual Studio code using the codetools of the Lazarus IDE.

1 Introduction

It is no secret that the code tools of the Lazarus IDE are excellent, and even surpass the ones in the Delphi IDE. So for coding in Object Pascal, the Lazarus IDE is an excellent choice.

But sometimes you need to code more than just Pascal. You may wish to edit Markdown, HTML, CSS, C or create Makefiles or shell scripts. This can also be done in the Lazarus editor, but then the support the editor offers you on top of basic editing is very limited: in the best case, you have syntax highlighting.

If you want more than that, you need to open another editor to do the editing. Many modern editors offer support for many languages: not only syntax highlighting, but also more advanced features one expects in an editor: code completion, identifier completion (Intellisense) finding references to a symbol, refactorings such as renaming a symbol and so on.

One such editor is Visual Studio Code (an evolution of the now defunct Atom editor):

`https://code.visualstudio.com/`

It has become very popular, and has a staggering amount of extensions - including several for Pascal.

The Visual Studio Code editor is managed by Microsoft, and Microsoft has introduced a standard for extending its editor with support for new Languages: The Language Server Protocol:

`https://microsoft.github.io/language-server-protocol/`

This standard has been adopted by several other editors, including Emacs, Vim, Delphi, Sublime Text, IntelliJ and the KDE editor suite (Kate & KDevelop). A more complete list is available here:

`https://microsoft.github.io/language-server-protocol/implementors/tools/`

Unfortunately, the Lazarus IDE is not in this list, as it does not yet support the LSP protocol: it would enable to use any language in the Lazarus IDE.

If the mountain will not come to Mohammed, Mohammed must go to the mountain: Pending support for the LSP protocol in Lazarus, the lazarus code tools can be used to implement the LSP protocol and extend other editors with first-class Pascal support.

Several LSP implementations using the Lazarus codetools are available on Github, but in this article we'll concentrate on one:

`https://github.com/genericptr/pascal-language-server`

2 The LSP protocol

The Language Server Protocol is based on a JSON-RPC communication mechanism. The editor starts a program that acts as a Language Server, and sends JSON-RPC messages to the process over standard input. It reads the results and possible commands from the LSP server from standard output.

This exchange looks as follows. The editor sends a request:

```
{
  "jsonrpc" : "2.0",
  "method" : "textDocument/didOpen",
  "params" : {
     "textDocument" : {
        "uri" : "file:///home/michael/source/testio/testio.lpr",
        "languageId" : "pascal",
        "version" : 1,
        "text" : "program testio;\n\n ... end.\n"
        }
  }
}
```

In this case the server replies with a command:

```
{
   "jsonrpc" : "2.0",
   "method" : "textDocument/publishDiagnostics",
   "params" : {
      "diagnostics" : [],
      "uri" : "file:///home/michael/source/testio/testio.lpr"
   }
}
```

The full list of commands a server can implement is documented in the LSP protocol (see the URL above). When the editor starts the server, a handshake is performed: the `initialize` command. In the `initialize` command (the first command the editor sends to the server), the client (the editor) indicates the capabilities it has, and the language server replies with the capabilities it has: this is normally a list of 'Providers' of certain functionalities.

This handshake is important, because not all servers support all commands, and not all clients support all commands.

A server can also specify custom commands: these are commands that are not part of the LSP Protocol, but for which the LSP protocol has a special command in place, aptly named 'executeCommand'.

The Pascal Language Server mentioned above implements various of the functionalities expected by the language server protocol:

textDocument/declaration Goto declaration

textDocument/implementation Goto implementation

textDocument/references find references to a symbol

textDocument/signatureHelp Show function or method signature (parameters).

textDocument/documentSymbol List of symbols in a project.

textDocument/documentHighlight Similar to `textDocument/references` find references to a symbol.

textDocument/completion Identifier completion

textDocument/hover Smart hints about your code

window/showMessage allow the LSP server to show a message in the editor.

workspace/symbol List all symbols matching a piece of text.

workspace/executeCommand Execute a custom command.

diagnostics Allows the server to send a list of diagnostics to the client: these can be warnings, errors etc. They are displayed in the editor (under 'Problems' in VS Code)

Additionally, the Pascal Language Server implements some custom commands:

pasls.completeCode Code completion: will complete the current class, define a variable etc. The equivalent of code completion in the IDE.

pasls.formatCode calls the Jedi code formatter on the pascal file.

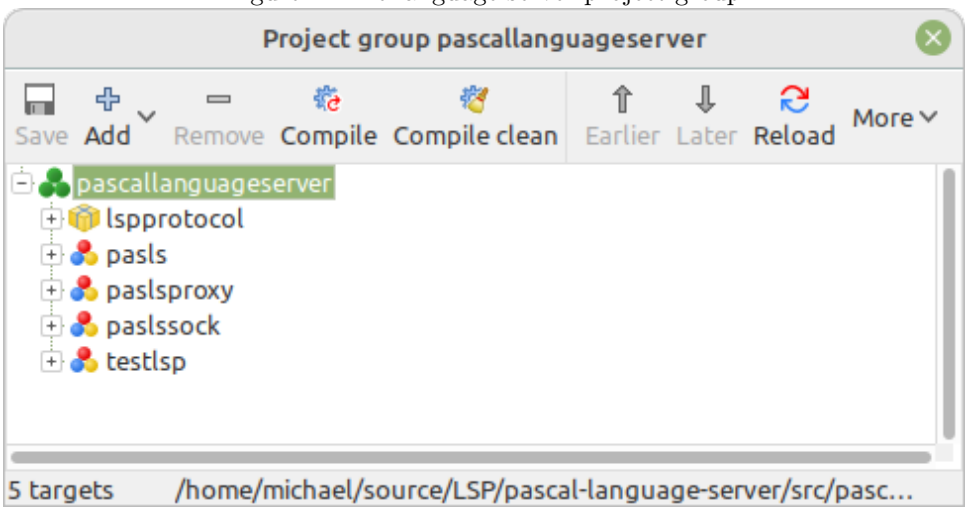
pasls.invertAssignment a refactoring which inverts the assignment statements in the selection.

pasls.removeEmptyMethods a refactoring which removes all empty methods from the current file.

More custom methods are being added to the server: theoretically, all code tools offered by Lazarus can be implemented.

What does the language server not do ? It does not compile the pascal code, it also does not offer functionality to edit form files. There are also several commands in the LSP that it does not implement. It also does not do syntax highlighting. (The 'pascal magic' extension in the VS extension marketplace does this for you).

Figure 1: The language server project group



3 Using the LSP server

To use the LSP server in VS Code, 2 things are needed:

1. Compile the LSP server.
2. Install an extension in VS Code that registers the LSP server, and the extra commands made available by it.

To compile the LSP server, you can clone the official repository from the URL above.

When you do so, below the `Src` directory, you'll have a project group file pascallanguageserver.lpg with 4 projects and a lazarus package (see figure 1 on page 4):

lspprotocol.lpk A package with the units that make up the language server protocol. It depends on the Lazarus codetools package.

pasls.lpi The actual LSP server program. This is the program that you must compile and use. In order to compile it, you must first open and compile the lspprotocol package.

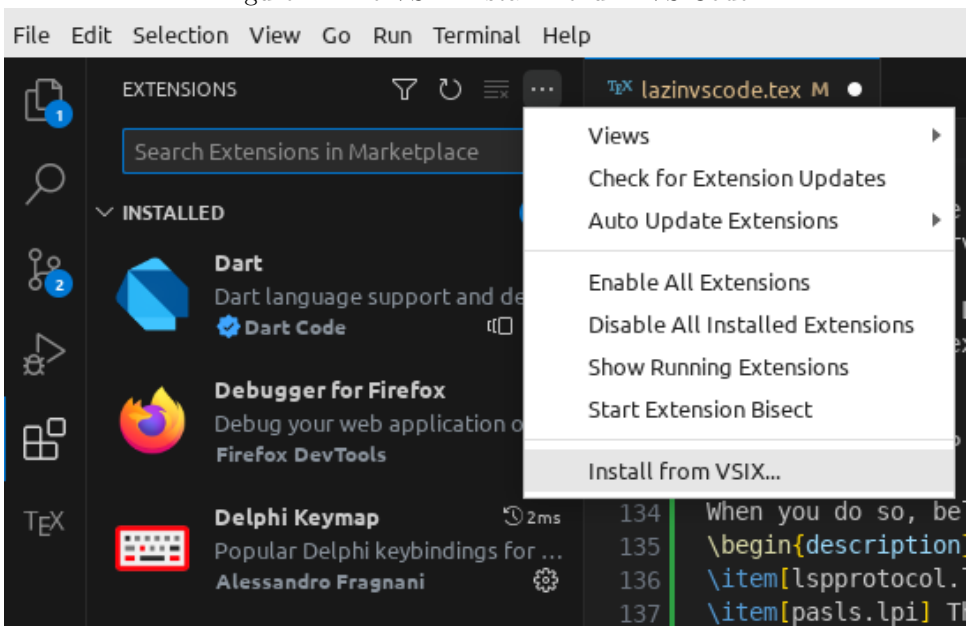
paslsproxy.lpi A proxy program that implements the JSON-RPC protocol on standard input/output and forwards the messages on a TCP/IP socket using a special high-speed message scheme.

paslssock.lpi A language server program that listens on a TCP/IP socket and implements the JSON-RPC protocol using the same message scheme as paslsproxy.

testlsp.lpi A minimal unit test program.

The paslsproxy and paslssock programs are only used for debugging the language process server: because the editor starts the LSP process, it is difficult to debug startup and message flow. By running `paslssock` in the Lazarus debugger and letting the editor start the `paslsproxy` program, you can debug the language server process. But for regular use, you only need the `pasls` program, and this is the one you should compile. It will compile on all platforms that Lazarus supports. You can

Figure 2: The VSIX install menu in VS Code



compile it with the released version of Free Pascal (3.2.2) or with the development version from git. If you do the latter, you'll have better syntax checking due to improvements in the pas2js parser (which is used to do syntax checking).

To create the pasls binary, open the project in the lazarus IDE and hit the compile key combination or use the Run-Compile menu item in lazarus.

You can also try to compile the pasls binary without Lazarus installed. You can checkout the lazarus sources from the git repository at

`https://gitlab.com/freepascal.org/lazarus/lazarus`

If you do this, you have to specify the paths to the lazarus codetools package and all other packages on which the latter depends:

codetools (components/codetools)

jcfbase (components/jcf2) - the jedi code formatter.

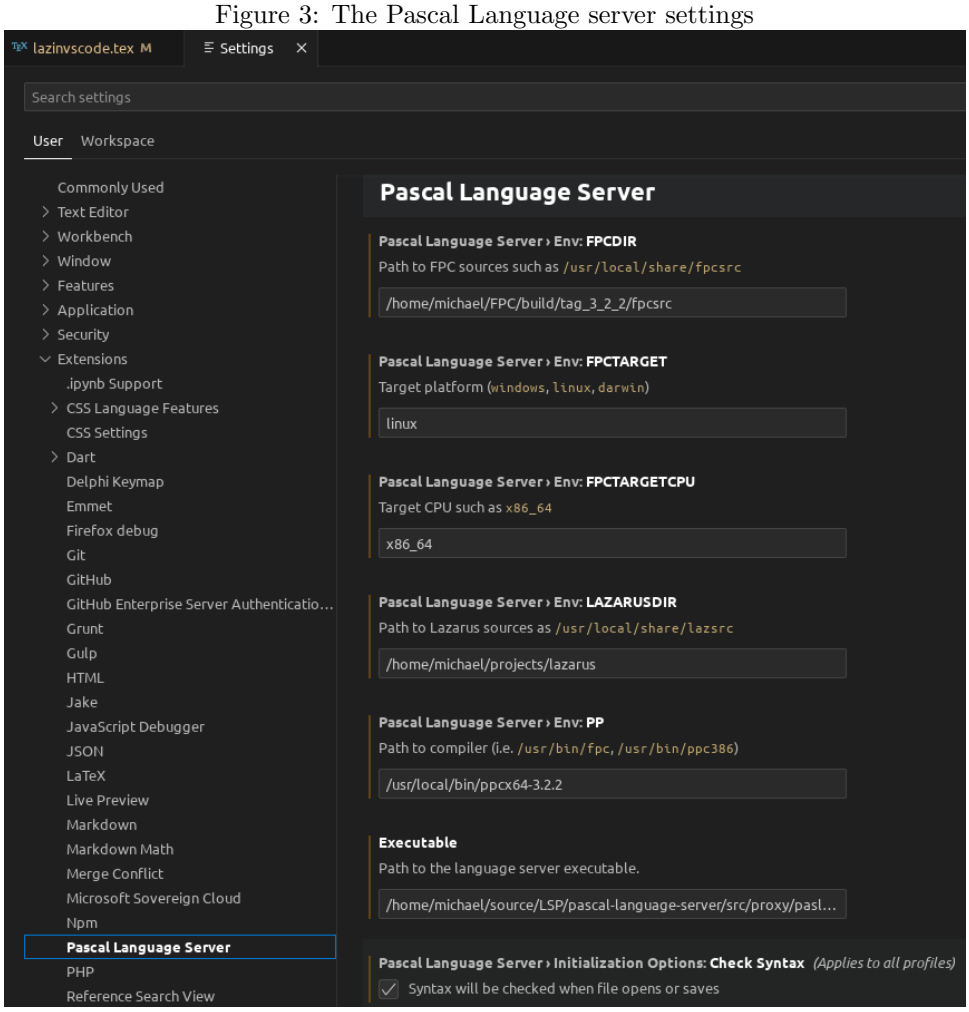
lazutils (components/lazutils)

Once you have a pasls binary, it can be used in an editor. For VS Code, there is an extension package available on github:

`https://github.com/genericptr/pasls-vscode`

You can package and install the extension in VS Code yourself, but a .vsix package file is available: this is an extension package for Visual Studio code, which can be installed using a menu item: in the extensions tab on the left of the IDE, the menu at the top contains an item 'Install from VSIX' (see figure 2 on page 5). You can use that to select and install the .vsix file. Once installed, there are settings in the VS Code ide that must be configured, see figure 3 on page 6. Important settings that are needed to make the codetools work correctly are the following:

Figure 3: The Pascal Language server settings



Env:FPCDIR The directory where the Free Pascal sources are located.

Env:FPCTARGET The target operating system.

Env:FPCTARGETCPU The target CPU.

Env:LAZARUSDIR The directory where the Lazarus sources are located (only needed if you work on programs that use some lazarus packages)

Env:PP The path to the Free Pascal compiler binary: the codetools use this to get some compiler information.

Executable The path to the pasls binary that you compiled.

format Config The path to the configuration file for the code formatter. This file can be copied from the settings file of a Lazarus installation (the file is called jcfsettings.cfg).

Furthermore, there are options that control the behaviour of the language server itself, they are part of the initialization options:

Check syntax When checked, the language server checks the syntax of the current file whenever you save it.

Document symbols When checked, querying for document symbols is possible.

FPC options these are options that you would normally specify when compiling your project: the codetools analyse these options to determine defines etc.

Include workspace folders as include paths When checked, all subdirectories of the VS Code workspace directory will be used as include paths (the `-Fi` command-line option of the compiler).

Include workspace folders as unit paths When checked, all subdirectories of the VS Code workspace directory will be used as include paths (the `-Fu` command-line option of the compiler).

Insert completion procedure brackets when checked and you complete a procedure call, the procedure call will have () brackets appended (even if no parameters are expected).

Insert completions as snippets when checked and you complete a procedure call, the procedure call is inserted as a snippet: it will have a cursor placeholder for the parameter (i.e. `'($0)'`).

Maximum completions the maximum amount of possible completions to be shown.

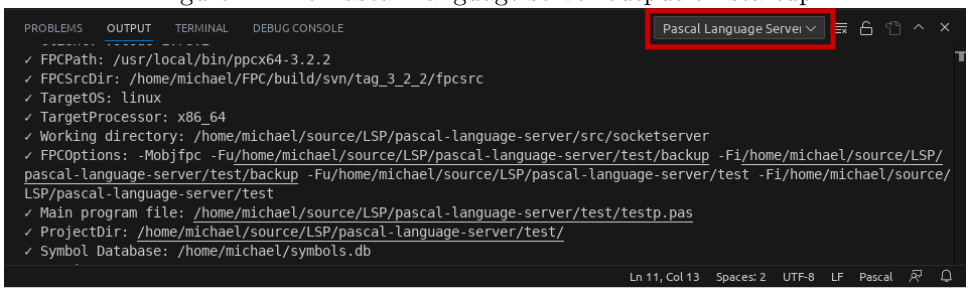
Minimalistic completions Provide minimal completion information : only the symbol name is shown, not what kind of symbol it is.

Overload policy determines how overloads are handled in the symbol list. A numerical value with the following meanings:

- 1 - Duplicate function names appear in the list
- 2 - Ignore overloads, only the first is used.
- 3 - Add a suffix which denotes the overload count

Program The main program file: This is used by the codetools to determine what units are part of the project, and to find references.

Figure 4: The Pascal Language server output on startup



Publish diagnostics When the codetools return an error in the language server, this error is reported as diagnostics.

Show syntax errors In case of syntax errors during a syntax error check, they are shown as small windows in the editor.

Symbol database a sqlite database to use for symbols: this database will be created and filled with symbols. This is then used as a cache: instead of parsing the files, the contents of the cache is shown instead.

Once you're done with the settings, you're all good to go. When you open a pascal project in VS Code, you can see that the pascal language server is correctly started in the output window (figure 4 on page 8), when you select the 'Pascal Language server' tool (see image, the red rectangle at the top right).

If the pascal language server was initialized correctly, you can start enjoying enhanced coding editing such as in figure 5 on page 9, in VS Code.

4 Executing custom commands

To execute the code formatter, you invoke the usual code formatting request in VS Code: the extension redirects this request to the language server. The standard key combination for this is `ctrl-shift-i`.

The code formatter uses a configuration file, you can set the location of the configuration file in the settings. The configuration file is an XML file. A sample file has been included in the github repository of the pascal language server. Most settings are self-explanatory, so editing the XML is possible, but at this time, the configuration file is most easily edited in the Lazarus 'Tools - Options' dialog.

The code completion feature of Lazarus is mapped to the standard key combination of Lazarus : `ctrl-shift-c` But you can simply type 'code completion' in the command palette and activate it like that.

To execute the other commands (remove empty methods or code formatter), you invoke the command palette (ctrl-shift-;) and type the description of the command, for instance 'remove' will result in a list as shown in figure 6 on page 9) in a later version of the language server, these will be added to the refactoring menu.

5 Conclusion

The Pascal Language server allows you to enjoy part of the tools that the Lazarus IDE offers you right in VS Code. The Pascal language server is not limited to use

Figure 5: A helpful hint about the function you're about to call.

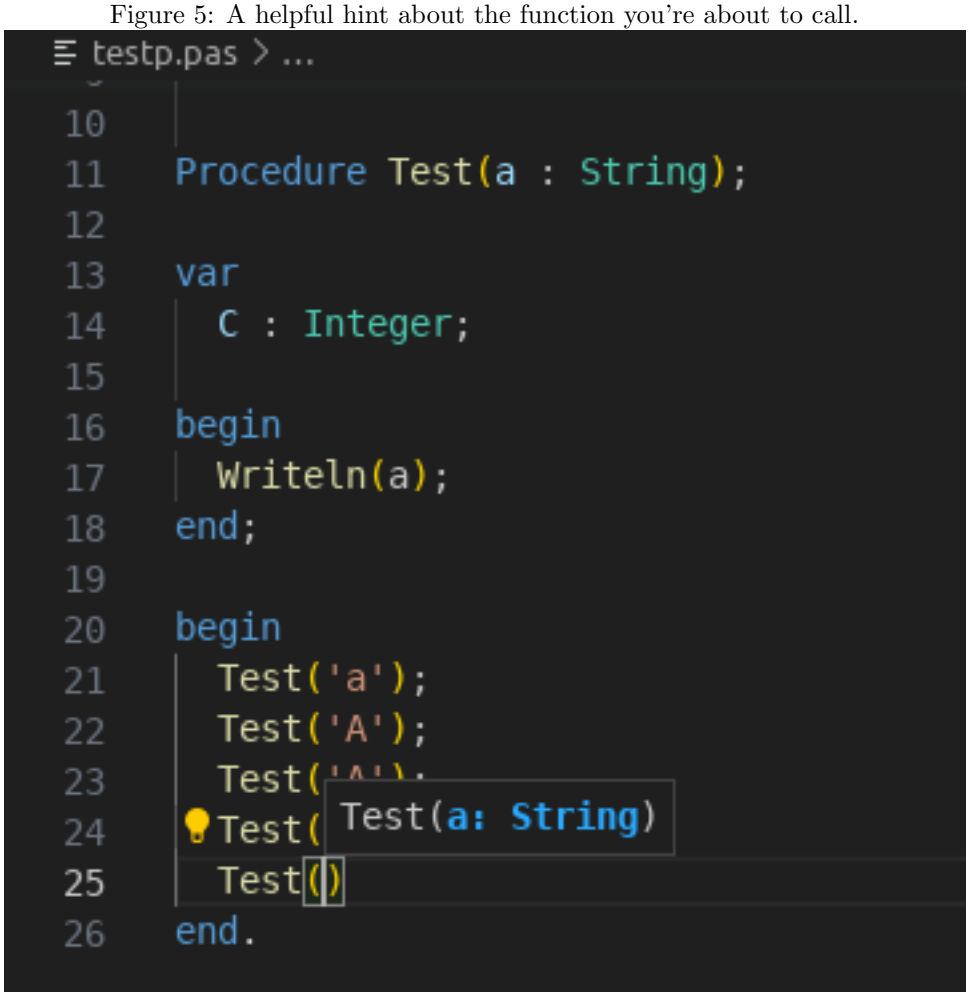
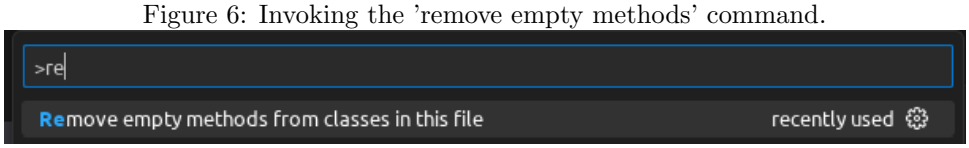


Figure 6: Invoking the 'remove empty methods' command.



in VS Code. It can be used with all editors that support the LSP process: one of the authors uses Sublime Text to develop Pascal. The pascal language server is under active development, so more goodies from the Lazarus code tools will be made available in the near future.