Language Server Protocol Support for the Amy Language

Compiler Construction 2020 Final Report

Raoul Gerber, Szabina Horváth-Mikulas, Eloi Garandel, Erik Wengle

EPFL

erik.wengle@epfl.ch, eloi.garandel@epfl.ch, raoul.gerber@epfl.ch, szabina.horvath-mikulas@epfl.ch

1. Introduction

In the first part of the compiler project, we have implemented an Amy compiler consisting of several modules:

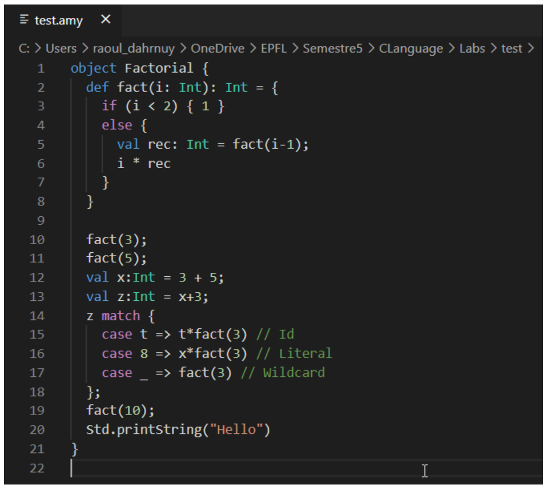
- The *Lexer* which converts a sequence of characters into a sequence of tokens
- The *Parser*, which converts these tokens into an abstract syntax tree *(AST)*.
- The *Name Analyzer*, already provided to us, which translates nominal trees into symbolic ones to verify that identical names refer to the same underlying entity.
- The *Type Checker*, which verifies that every statement follows the given type rules (no errors based on the kind or shape of values that the program is manipulating).
- The *Code Generator*, which converts the AST into WebAssembly.

Outside of these models we have also written an interpreter from Amy to Scala using a front-end which was already provided to us.

Our extension acts as an addition to what we have already implemented, requiring only a few changes in the original compiler. The server will run the compiling pipeline, except for the code generation, in order to be able to provide feedback such as type errors or missing declarations to the user.

2. Examples

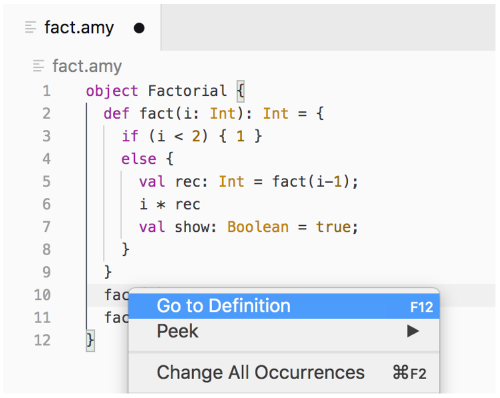
Our extension aims to provide information useful to the programmer, such as a code coloring, and services, such as auto-completion. They may lower the workload of the programmer, and can be applied anywhere in the code while writing an Amy program.


test.amy
C: > Users > raoul_dahrnuy > OneDrive > EPFL > Semestre5 > CLanguage > Labs > test >

```
object Factorial {
  def fact(i: Int): Int = {
    if (i < 2) { 1 }
    else {
      val rec: Int = fact(i-1);
      i * rec
    }
  }

  fact(3);
  fact(5);
  val x:Int = 3 + 5;
  val z:Int = x+3;
  z match {
    case t => t*fact(3) // Id
    case 8 => x*fact(3) // Literal
    case _ => fact(3) // Wildcard
  };
  fact(10);
  Std.printString("Hello")
}
```

Figure 1. An example of coloring



```
object Factorial {
  def fact(i: Int): Int = {
    if (i < 2) { 1 }
    else {
      val rec: Int = fact(i-1);
      i * rec
      val show: Boolean = true;
    }
  }
  fac
  fac
}
```

Figure 2. An example of go to definition

Another example is the *go to definition* feature, which lets the programmer right click a definition, type or function signature, and then gives them the option to move directly to the file and line of code where the code in question was initially defined.

Regardless of the cursor position along the word, the entire identifier will be sent to the language server, which will search the abstract syntax tree for the definition and its position and return it to the client.

Such features have been implemented for a large range of programming languages like Python, Scala, Java, etc.

3. Implementation

In order to provide the support we described in the previous two chapters, the IDE and programming language communicate with each other using a Language Server Protocol *(LSP)* standardized by Microsoft. It enables developers to develop one LSP per language, regardless of the IDE. A Java implementation of the LSP protocol is called *LSP4J*. We added this library to our server.

3.1 Theoretical Background

The protocol works as follows:

- The IDE acts as the language client. It sends requests (asking for the cursor's position, the file name or to perform a given action) to the server and receives data from it to display.
- The programming language acts as the server. It receives requests from the client, resolves them, and send back data such as errors, warnings, positions or location of a given definition, etc.

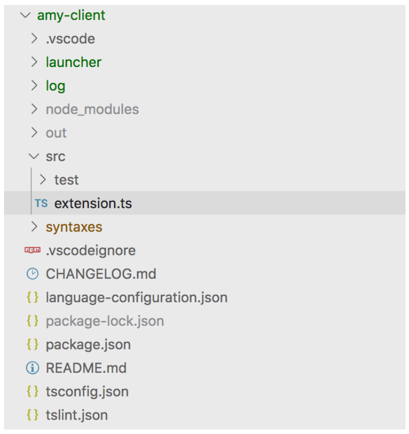
Those requests are transmitted using JSON-RPC, a remote procedure call protocol encoded in JSON. On the client as well as at the server side we used libraries that implement the LSP4J protocol

3.2 Implementation Details

3.2.1 Server-Client implementation

We generated the basic layout of our TypeScript based plugin, named `extension.ts` for Visual Studio Code *(hereinafter referred to as VS Code)* using a tool called Yeoman. The plugin gets activated by a command registered in a `package.json` file - whenever one opens a `.Amy` file, our extension is activated.

Using a language client NPM module available for VS Code, we can import our language server protocol, instantiate our server and establish a connection between VS Code and our server.



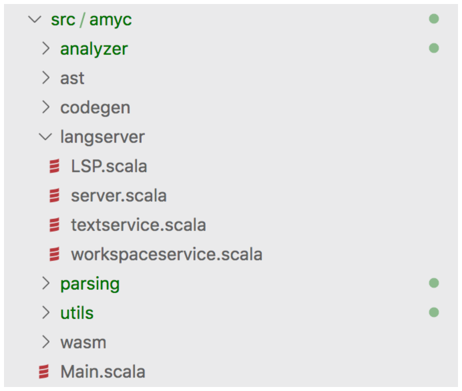
```
amy-client
  .vscode
  launcher
  log
  node_modules
  out
  src
    test
    extension.ts
  syntaxes
  .vscodeignore
  CHANGELOG.md
  language-configuration.json
  package-lock.json
  package.json
  README.md
  tsconfig.json
  tslint.json
```

Figure 3. Architecture of the VS Code client

LSP4J has the interfaces for LSP. We implemented them to support our language. There are three main interfaces, i.e. Language Server, TextDocumentService and WorkspaceService. Their implementations are AmyLanguageServer (server.scala), AmyTextDocumentService (textservice.scala) and AmyWorkspaceService (workspaceservice.scala), respectively.

Language server is the initial class, which initializes the language server for TextDocumentService and Workspace Service. The former provides the language feature interface while the latter the workspace feature interface.

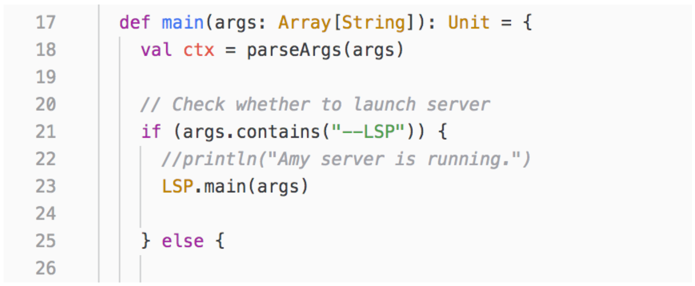
There is another class in LSP4J called Launcher. The launcher is what allows the client to connect to the language server. The launcher gets input and output streams as arguments so the IDE can write the request into the input stream and Language server can write the result into the output stream. This will be called by the VS Code plugin to get the server running. The main entry point to the server is in LSP.scala.



```
src / amyc
  analyzer
  ast
  codegen
  langserver
    LSP.scala
    server.scala
    textservice.scala
    workspaceservice.scala
  parsing
  utils
  wasm
  Main.scala
```

Figure 4. Including the language server to the Amy compiler under langserver folder

In order to use our extension, we added a separate argument, namely `--LSP`, to run the Amy compiler in server mode. The main entry point to our server is the Main.scala file. We created a fat jar from our compiler with the integrated server and included it to the VS Code extension.



```
def main(args: Array[String]): Unit = {
  val ctx = parseArgs(args)

  // Check whether to launch server
  if (args.contains("--LSP")) {
    //println("Amy server is running.")
    LSP.main(args)

  } else {

```

Figure 5. Running the compiler in server mode by LSP flag

3.2.2 Features Implementation

The LSP4J text services for Amy, i.e. what our language server should respond to the requests, were implemented using Scala. Of all the available text services, we chose to provide a *go to definition* option. Each time the client sends such a request, the server runs the compiling pipeline up to the stage of the name analyzer in order to obtain the abstract syntax tree,

Then the first step is, given the position of the cursor converted to an *AmyPosition*, to find the associate identifier to this position. We do so by recursively iterating through the Abstract Syntax Tree of Expressions and looking if for one given expression, its position range contains the given position. If we did not find any correspondence in the Tree, we search into every Class and Function definition.

To be able to achieve the described AST search, we had to introduce an *endPosition* in the *Positioned class* : it represents the end position of a given token in the code. Thus, when using the *go to definition* feature, one could right click on any part of the keyword and still get its definition. Otherwise, without this range property, one can only click on the start of the keyword to get a correct result.

At this point, if no correspondence was found, then the *go to definition* feature returns an empty output.

In the case we found an identifier, we search a second time in the tree to look for local variables and in the Class/Function definitions to get the position of the original declaration of the given identifier. This time we are comparing identifiers instead of positions. Then, the features will output a new position which represents the declaration of the wanted value.

The syntax highlight feature is much simpler. We defined a *.json* file, *amy.tmLangauge.json*, which contains every syntax pattern that we want to match to a wanted color. Once we launch the extension, when a pattern is recognized, it will be colored accordingly.

3.3 Current Limitations

There are currently 3 main limitations to what we've achieved for *go to definition*:

- Our way to trim the file URI provided to a usable version relies on it having /Users/ at a top-level. This is fine for default Windows and MacOS, but probably not for Linux distributions.
- To use certain features, the file needs to be in a compilable state. Our *go to definition* feature uses the identifiers generated during the NameAnalyzer phase. But when a file isn't fully compiled, it crashes and hence, we do not get correct identifiers.
- While we can handle several files, several modules inside a file and several definitions inside a module, the files being searched need to be hardcoded in the server.

4. Possible Extensions

There are plenty of text services that were not implemented, such as auto-completion, intermediate type checking, etc. We set our focus on the *go to definition* service in order to have a proper understanding of the entire communication process between client and server.

Furthermore, one can make the server more efficient by not having it compile with every single request, but only on specific changes, or have it cache certain parts of the procedure in order to reduce the overall recompilation time during the programming process.

In the end, possibilities are infinite. The purpose of an universal protocol is to let the programmer find which features are the most appropriate to him/her and then, thanks to the standardization of this protocol, can implement it easily in order to get a tool that is more useful.

5. References

Ballerina : programming language, *https://ballerina.io/*

Yeoman : The web's scaffolding tool for modern webapps, *https://yeoman.io/*

Microsoft LSP repository *https://microsoft.github.io/language-server-protocol/*

Visual Studio Code LSP documentation, *https://code.visualstudio.com/api/language-extensions/language-server-extension-guide*

LSP4J : Java implementation of the language server protocol intended to be consumed by tools and language servers implemented in Java. *https://github.com/eclipse/lsp4j*