Modular Language Server: Design and Implementation

Innopolis University

Thesis submitted to The Innopolis University in conformity with the requirements for the degree of Bachelor of Science.

presented by

Mike Lubinets

supervised by

Eugene Zouev

12/20/2017

Modular Language Server: Design and Implementation

Contents

Abstract

abstract ...

Chapter 1

Introduction

Nowadays we have a lot of mature toolkits and integrated development environments for a set of widely used programming languages evolved through years to fulfill needs of the most of software development industry.

However, such software products are highly complex and often have a monolithic architecture, usually standalone from the language compiler infrastructure which makes it hard to replace key components or implement support for additional languages.

In this paper we will see how conventional compilers have been architectured and what complications this consequently brought to modern IDE implementations, review the modern approach to compiler construction and implement a flexible distributed integrated development environment with the Language Server Protocol, for a multi paradigm SLang[1] programming language.

Chapter 2

Literature Review

2.1 Conventional compilers in the modern world

Since the middle of 20^{th} century researchers and the industry have done a great work in compilers development focusing on the main goal of classic compilation problem: producing fast and effective object code for execution on a virtual machine or a microprocessor.

However, the other compilers‘ capability as developer‘s code inspection instruments able of providing comprehensive information of code semantics remains uncommon and rarely well-developed in the modern compilers of popular programming languages.

The most common illustration of this problem can be found in an average Java developer‘s set of instruments:

Java code is usually compiled with Sun Java Compiler. “Being a monolithic program, constructed as a ‘black box‘”[2], Sun Java Compiler can only accept the input code and produce optimized JVM byte code.

Yet a modern development environment includes a set of tools for programming assistance and requires advanced language syntax and semantics inspection, which is not possible without building a Semantic Representation[2]. To build it one needs to re-implement core functionality of a compiler.

It‘s easy to understand the very reason of the issue looking back to the past: traditionally programs have been considered as mere text objects to be converted into an executable code. According to this assumption, compilers were designed in a very logical way: they haven‘t maintained any semantic representation of source code, only some low-level internal IR.

These compilers‘ IRs have very limited set of use cases[2, 3], moreover they are good for the only task: object code generation for several microprocessor architectures. Also compiler‘s IR is not stable and tends to change very actively during compiler development[4]. Hence the internal compiler‘s IR can‘t really be used to build good and reliable development tools.

The situation is even more frustrating with C++ tooling: the language syntax and semantics is a lot more complicated than Java‘s, so building a custom compiler is a very complex task.

As a result, there is a notable lack of instruments for C++[3], and existing ones are pretty sophisticated: JetBrains CLion IDE implements its own parser and semantic analyzer to build autocompletion, refactoring and static analysis tools upon their own C++ SR. Being a complex software

product, CLion‘s parser tends to have its own misfeatures and a few month implementational lag to fully adapt for new standards.

Microsoft Visual Studio suffers from this too: VC++ generates IR that is useful only for code generation: it is fragmented and very low-level. The C&C++ IntelliSense tooling in Microsoft Visual Studio IDE is implemented as a solution separate from VC++ compiler.

2.2 Modern compilers and SR: a new hope

In spite of the fact that traditional compilers are widely used today, their lack of IDE integration capabilities were realized long ago and currently a lot of new languages are aiming to implement a Semantic Representation as a stable IR shared with external tools.

Following [2, 3], unlike a traditional compiler IR, Semantic Representation contains a full knowledge of a program, including the aspects that are implicit in the source code. This trait enables some pretty powerful opportunities based on semantics analysis:

- Code generation
- Distributed (or recursive) Validation
- Human understandable visualization
- Static analysis
- Program interpretation
- Semantic Search: the very powerful technique of querying code semantic objects (find all classes derived from class C that do not override the virtual function f)

There are two main ways to represent an SR and share it with SR clients: to provide an access API operating on an SR (proprietary) binary format [4, 5], relational database representing a software structure [6], or to output SR as an open textual format[7].

In accordance to [2],“Generally speaking, API is a universal way to implement any required functionality, however with changing requirements it‘s impossible to predict a spectrum of clients‘ needs”. And an open SP format can be a solution to potential problems: “open formats usually have a lot of access means: from simple APIs to high level specialized products. Besides, it‘s possible to implement one‘s own interfaces to process SR represented with open format”.

A particular format may be something self-designed[7] or a standardized solution such as XML[8], or JSON[9], as an alternative.

2.3 LSP and distributed approach to building development environment

Considering the things discussed above, nowadays we have a solid basis to provide a good tooling based on semantic analysis: methods to represent software source code‘s Semantic Representation and evaluate it accordingly to clients‘ needs.

Modern IDEs apply those methods to deliver a decent service, but still there is a problem: those software products use their own implementations of compilers, usually proprietary and unrelated to the original language‘s development team. It implies a set of problems noted in the 2.1.

Having a good modern compiler capable of generating an SR makes things a bit less complicated but still doesn‘t solve the language-specific IDE implementation time and cost problem.

Obviously, these problems are not unique for the IDE class of products, but for any big monolithic architecture, and the solution may be pretty straightforward: if we can lower the bonding of the system and represent a development environment as a set of tools instead of one integrated solution, we can distribute the IDE implementation to have a set of disjointed modules:

- An editor
- Compiler to SR
- SR clients (described in 2.1)
- Protocol between an editor and the language-specific part

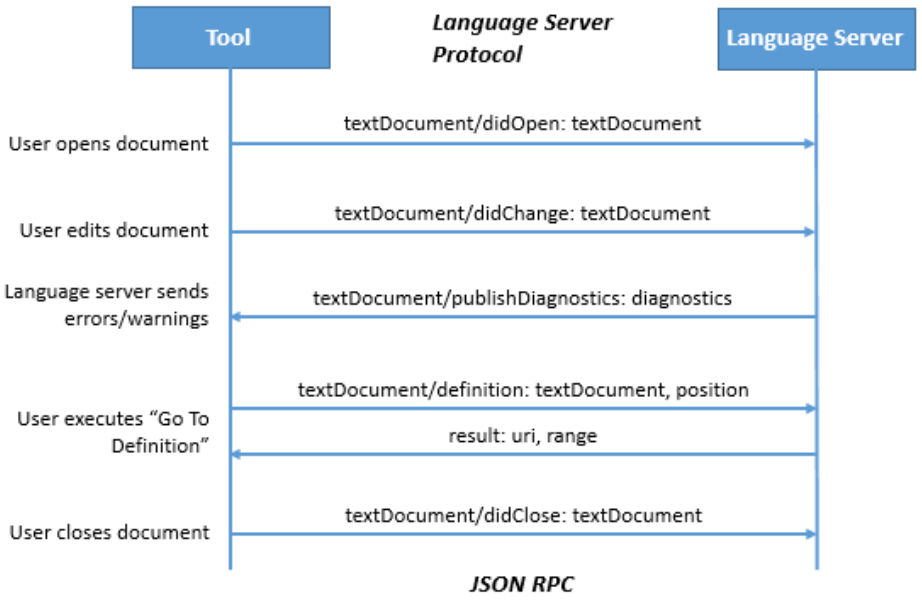

Figure 2.1: Language Server Protocol

The last one have not been introduced yet: the protocol to connect any third-party tool to the language infrastructure, achieving a decent IDE-like functionality without its maintenance and development costs

Language Server and Language Server Protocol introduced by Microsoft in 2016 represent a development environment as two disjoint parties:

- Language Servers to implement all the SR analysis things
- Clients as editors or other development tools using the LSP to communicate with Language Servers [10]

2.4 Conclusion

Conventional compilers with a monolithic architecture, that are only good at executable code generation, are hard to integrate into a modern development environment as they do not share semantic representation of the source code, thus to develop a good IDE one must write their own source code to Semantic Representation compiler.

A modern compiler (that does share a high-level intermediate representation) is a big step towards simpler language toolings and it can become even more convenient combined with a distributed IDE architecture that splits an editor and the language toolchain into two disjoint parts, linking them via a standardized protocol.

This approach gives language developers a great opportunity to make use of an existing development infrastructure, providing their Language Server for a giant set of development tools, as well as a way to fearlessly experiment with new and existing analysis techniques, e.g. a Software Knowledge Base[11], described by Bertrand Meyer, may be implemented as a language server module, as an alternative approach to the one selected by the original author in 1985: integration of analysis tool into an editor was not possible back then, but this is the exact thing that LSP is good for now.

Concluding, the Language Server may be considered to be the most feasible solution to rapidly bootstrap rich development infrastructure for aspiring new languages, with a broad path to evolve further.

Chapter 3

Methodology and architecture

3.1 Compiler integration

The Language Server idea is to launch the LS instance in the same project directory opened in the editor, and connect it to the editor via Language Server Protocol.

A Language Server is responsible for language-specific editor features, it works on the language Semantic Representation and other metadata to perform semantic analysis and consequently provide the editor with usable data in the agreed format via Language Server Protocol. As Language server heavily relies on the modern compiler, that exposes the SR, we need to implement a way to integrate compiler into the Language Server and to enable their inter-operation.

There are two possible ways to achieve that: either use the compiler as a library or invoke it in a separate process, feeding specific command line arguments.3.1

invoking as a command	**using compiler as a library**
simpler integration	harder integration
very limited invocation options	complex invocation strategies may be expressed
need to (de)serialize data	can exchange binary data
need to implement IR traversal in the LS	compiler can expose AST traversal API
need to describe compiler internal data types in the LS	compiler can expose internal data types

Table 3.1: Compiler integration methods comparison

Since the SLang[1] compiler does not expose any AST traversal API or internal data types, most of the traits specific to an “integration as a library” option will not be used in our case. Moreover, the compiler provides a stable JSON-formatted SR, which, being a text-serialized format, can be easily transferred via an operating system channel like standard output[12].

This way, we get a number of convenient features: the compiler can be invoked by our Language Server as a command call,this option is easier to implement on both Language Server and compiler sides. And since we are not limited with any functionality that would require “compiler as a library” traits, we can declare this way of integration the most feasible in our case and stick to it.3.1

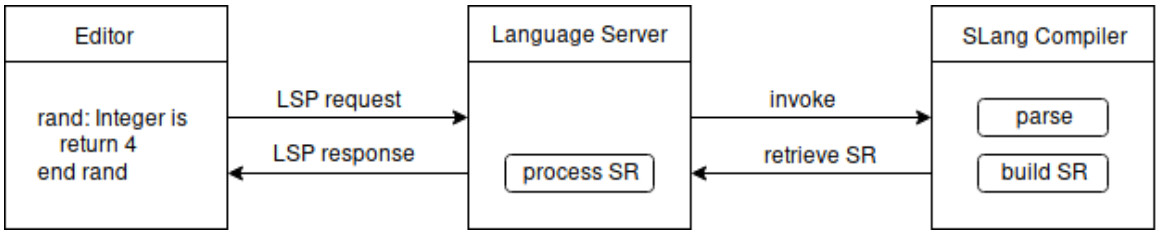

Figure 3.1: Workflow with Language Server and compiler

3.2 Language Server Extensible Architecture

The main idea of this research is to bring architecture of Language Servers to the next level, make it modular and extensible, thus allowing third parties to throw in additional functionality for the SLang tooling with no need to hack into the Language Server code.

Therefore, the architecture of the SLang Language Server is divided into two aggregate components: 3.2

- The Core
- Module System

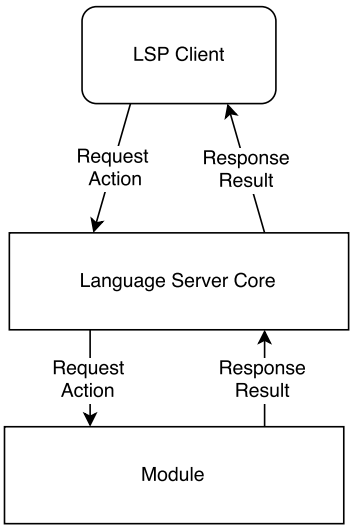

Figure 3.2: Language Server High-level Architecture

3.2.1 Language Server Core

Language Server Core is a basement level of the Language Server on which Language Server Modules will operate. Responsibilities of LS Core include:

- LS Client connection maintenance
- Module registry maintenance
- Routing of incoming requests and data control flow between modules

Each of these responsibilities we shall describe in detail below.

Client connection maintenance

According to the Language Server Protocol[10], the client controls the lifetime of a server, i.e starts it and shuts the server down on demand. After startup, the client connects to the server using one of the transports. Since the transport level is not constrained by the LSP, specific transport can vary in different implementations.

Language Server Core should support several types of transport and be able to operate on them to accept requests and respond to the client. The list of widely used transports we are going to implement is

- stdin/stdout
- TCP
- UDP

Implementing that list will supply the most of LSP clients with an option of how to work with the SLang Language Server.

Module Registry

To be a foundation for an extensible modular architecture Language Server Core needs to have a subsystem for module registering, maintenance and inter-operation organization.

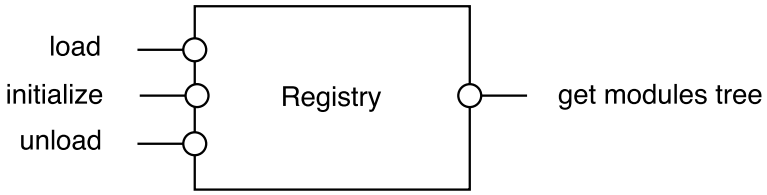

Figure 3.3: Module Registry API

The registry should register a module and share its status, as well as information on how to launch and connect internal (core) and external modules. On startup, the registry should initialize its state using a predefined directory containing configuration files. Afterwards, it should maintain an API to load and unload additional modules via LSP. Consequently, we need to extend the LSP with additional commands for the modules registry:3.3

- registryCtl/load: register the module.
- registryCtl/unload: unregister the module.
- registryCtl/status: query module status.

Dispatcher and data flow control

After startup, connection setup, module registering and initialization, the Language Server accepts the first request from the client. This request gets validated by the Language Server Core, and then, after looking up the Module Registry, the request gets handed over to the beginning of a processing pipeline responsible for handling this type of requests.

The dispatcher part is basically a glue, that connects all Language Server components together and maintains the data flow edges of the modules graph, enabling module inter-operation by the rules loaded into the Registry, that are discussed further in the section 3.2.2.

There is a simpler alternative approach to module inter-operation organization: let the modules send data to each other and organize pipeline as they want. Although this peer-to-peer schema here would save a lot of bandwidth, it would also inevitably lead to the dependency hell, as such an approach would require having every module knowing about each other and to be connected to each other. Thus, here we face the classic client/server trade-off: we can offload a "server" (LS Core) only if we complicate a "client" (modules). Since the client side is to be developed by third parties, the simpler it is – the better: the server, controlling all the data flow, will leave the module developer only with the business logic implementation tasks.

3.2.2 Module System

Language Server is a great idea that enables IDE-like functionality for comparably simple text editors, but currently these are mostly designed as a monolithic software, while their service functions are naturally extensible, e.g. it is common for a static analyzer to have modules for different diagnostics, and similarly one of Language Server functions is to provide an editor with diagnostics, so here we can have a hierarchy of at least two modules – the compiler diagnostics and the diagnostics provided by an external static analysis tool. We can go even further and implement a static analyzer as a combination of the base static analysis module with a bunch of atomic diagnostic modules derived from this base module. The same logic is applicable to every Language Server service to some extent.

Separation of the monolithic core from the actual functionality implemented through modules will significantly lower the Language Server internal bonding, and will make module implementation relying on as stable API as possible, therefore allowing the core and modules development to be performed simultaneously with no mutual API breakages.

Therefore another benefit of an extensible architecture that we can derive is the Language Server easy adoption for any specific corporate needs. As implementing some additional functionality would be as easy as developing a plug-in using the Language Server Modules API, corporate users can extend the Language Server according to their specific requirements, i.e add custom diagnostics, enforce code style, or even hook-up the proprietary static analysis or code generation tools.

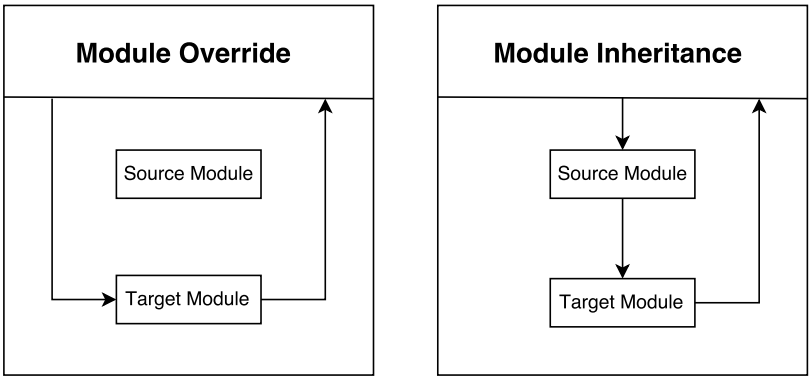

Figure 3.4: Module Hierarchy

One of the ways to develop such architecture has already been introduced above: the hierarchy of modules expressed with the basic Object-Oriented Programming terms3.4. In this hierarchy, modules can either override other modules or extend them in a way of post-processing the results of the base module computation. So the modules would form a classic tree, deriving from a base module, and intercepting the data flow. The example of such a module hierarchy is illustrated in the figure 3.5

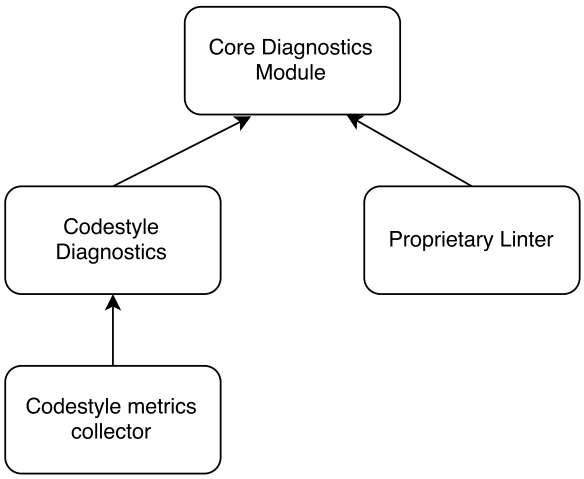

Figure 3.5: Example Modules Tree

We already mentioned the hierarchy of the base and derived modules above. Taking the modern Object-Oriented languages‘ type systems as an example, we can build a system with the base mod-

ules, which are to be deployed bundled with the Language Server and responsible for one particular LS Protocol method each. These modules will mainly provide the basic runtime for all the derived modules:

- set up data types: the input information, analysis context, and the results.
- perform the construction of the context (if applicable).
- intercept analysis results of the derived module and send them to the client through LSP.

As one could have figured, it is not really a conventional O-O paradigm inheritance model: core modules do not let the derived ones to override their logic as they work with the Language Server Client and therefore are forced to live within the same process with the Language Server due to resources sharing. The process of a modular Language Server operation is showed on the sequence diagram 3.6.

Hereby, we have two types of modules:

- Core Modules: embedded into the LS process, exposing the run-time for external modules, cannot be overridden.
- External Modules: pluggable things, derived from the core modules, can be overridden.

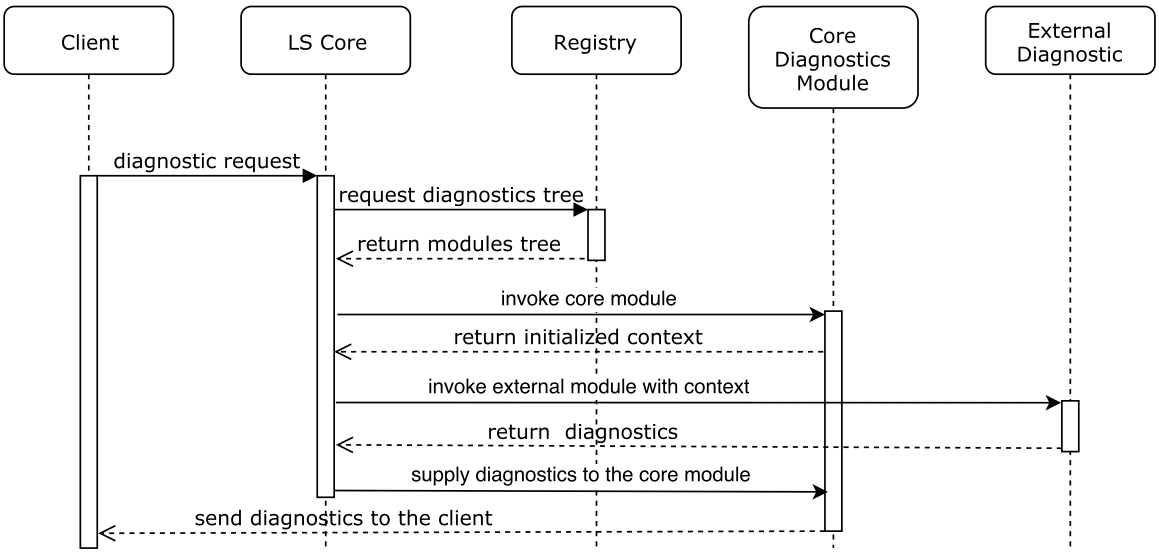

Figure 3.6: Modules invocation sequence diagram

Summing up, now we have a way to integrate the compiler into the Language Server, powerful extensible architecture that allows to throw in additional functionality for the Language Server, and modules hierarchy to start designing the core modules for the Language Server Protocol methods, as well as the external ones to perform actual analysis and supply Language Server Clients with the source code insights.

3.3 Development Plan

To implement a complex system it is vital to split the work into phases and introduce the iterative development plan, that would cover a small self-contained part of work on each iteration, in such a way that the result of each iteration could be presented as a working product.

Also, each iteration should include unit and integration testing to form good test coverage at the end of the development and to simplify the process of project evolution through use of regression testing.

A Dummy Language Server
At first, the very basic functionality should be implemented: accepting the connection and handling a request with a hardcoded response

SR-driven Language Server
The next step would be to integrate the compiler and the language Semantic Representation to feature "jump to definition" for the SLang Language Server.

Basic Modular Language Server
After all the groundwork with the protocol and compiler inter-operation, modular architecture implementation may be started: without foreign process extensions for now, including only built-in modules, that operate in the same address space with the Language Server, and the simple registry for them, to mock the basement for the future extensible LS.

Extensible Language Server
Finally, the last step is to extend the registry so it would handle configuration files and load external modules and to extend the inner protocol and API to work with modules that are operating in external processes, thus completing the extensible Language Server implementation for SLang. The final architecture after this stage is illustrated in the figure 3.7

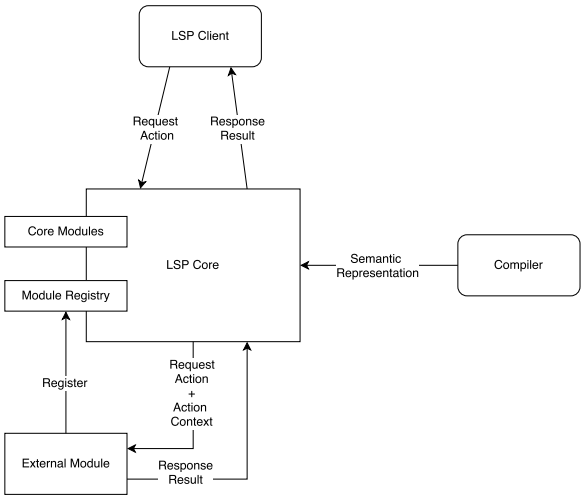

Figure 3.7: Final architecture

3.4 Approaches to implementations

The most of the Language Server functionality heavily relies on the language‘s semantic representation. For the first prototype we are going to be using the SLang semantic representation and will build a few tools for it.

The SLang is using a json-formated SR with an object hierarchy described in the following section.

3.4.1 SLang Semantic Representation format

Slang has three scoping components: Unit, Block and Routine, where any nesting order is valid. Each of these components has context with encapsulated members as an array to keep the consistent order:

```
{
    "type": "Unit",
    "context": [
        {
            "type": "Routine",
            "context": [
                "type": "Block",
                "context": [
                    ...
                ]
            ]
        }
    ]
}
```

In addition to the described basic scoping hierarchy, SLang JSON SR unilizes some extra fields with metadata and semantics extracted from the source code:

- **name**: the program-specified name of a specific program entity (variable, routine, etc.), optional.
- **id**: unique entity identifier, used for SR internal referencing.
- **ref**: internal SR reference to an **id**, used for referencing the parent unit, variable definitions, etc.
- **span**: text locator of the entity containing the start and the end positions represented as a line:column pair.
- **signature**: the routine special case field containing the routine signature as input and output contexts.

The meaning of each of these fields is implementation-defined and their semantics rely on the entity type they are contained within. For instance, for a Unit type, the ref field will be pointing to a parent Unit, and for a Var within an Expr entity the ref field will be a reference to this particular

variable definition. Thus we can avoid lengthy SR leaf definition in the Language Server code saving all the important semantics.

3.4.2 Jump to definition

The most basic developer tool as well as the most basic toolset part is a jump to definition command. Basically it helps the developer to instantly open the file and the line with the definition of some code entity used in the source code.

To implement this part of the Language Server via analysis of SR, only three of its components are necessary:

- **id**: every definition inherently has its own unique identifier.
- **ref**: every usage of an entity defined somewhere references the original definition by its id.
- **span**: knowing the id, the location of the definition can be found.

3.4.3 Code completion

Another very common instrument in a developer toolbox is autocompletion, provided by an IDE. In our case the provision of an autocompletion is on the Language Server.

The most usual case of autocompletion usage is navigation among the encapsulated class (or Unit) members, where autocompletion can provide user with the useful suggestions on the names of these members and their types. Using the recursive jump to definition search, we can find a scope with the members to be suggested, read theit metadata and present it to a user in a readable format.

Chapter 4

Implementation

4.1 Language Server Design

Design decisions behind language Server, its architecture

4.2 Language Server Implementation

4.2.1 Compiler inter-operation

Implementation of compiler inter-operation in language server

4.2.2 LSP implementation

Implementation of Language Server Protocol

4.3 LS Modules

Description of basic Language Server module design and implementation

4.3.1 Semantic Based Highlights

Description of the semantic based highlights

4.3.2 Autocomplete

Description if autocomplete implementation

4.3.3 Documentation Generator

Description of documentation generation implementation

4.4 Language Server control utility library

Description of C ABI LS control library API

Chapter 5

Evaluation and Discussion

Recap of thesis subject

LSP standard complaisance testing

Performance testing

Estimation of developement cost of the full-featured IDE for SLang, comparing that with time and cost of Language Server Implementation

Discussion of future development opportunities

...

Chapter 6

Conclusion

Recap

Concluding, the Language Server may be considered to be the most feasible solution to rapidly bootstrap rich development infrastructure for aspiring new languages, with a broad path to evolve further.

...

Bibliography

[1] Eugene Zouev and Alexy Kanatov. “SLang Programming Language”. 2017.

[2] Eugene Zouev. “Evolution of Compiler Architecture”. In: *ISBN 5-317-01413-1* (2005), pp. 322–331.

[3] Eugene Zouev. “Semantic APIs for Programming Languages”. In: (2010).

[4] Free Software Foundation. *Ada Compiler GNAT*. 2016. URL: http://www.adacore.com/home/products/gnatpro/ (visited on 09/30/2017).

[5] Aemon Cannon. *Building an IDE with the Scala Presentation Compiler*. URL: http://ensime.blogspot.ru/2010/08/building-ide-with-scala-presentation.html.

[6] Mark Linton. *Queries and Views of Programs Using a Relational Database System*. 1983.

[7] The Rust Team. *Rust Programming Language Blog: MIR*. 2016. URL: https://blog.rust-lang.org/2016/04/19/MIR.html (visited on 09/30/2017).

[8] R. Germon. *Using xml as an intermediate form for compiler development*.

[9] ECMA-404. “The JSON Data Interchange Format”. In: *ECMA International* 1st Editio.October (2013), p. 8. ISSN: 2070-1721. DOI: 10.17487/rfc7158. arXiv: arXiv:1011.1669v3. URL: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf.

[10] Sourcegraph. *A community-driven source of knowledge for Language Server Protocol implementations*. URL: http://langserver.org/ (visited on 09/30/2017).

[11] Yuan Wanghong et al. “C++ program information database for analysis tools”. In: *Proceedings Technology of Object-Oriented Languages. TOOLS 27 (Cat. No.98EX224)* (), pp. 173–180. DOI: 10.1109/TOOLS.1998.713598. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=713598.

[12] The Open Group. *Standard I/O Streams*. 1997. URL: http://pubs.opengroup.org/onlinepubs/7908799/xsh/stdio.html.