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

Оглавление

Аннотация

На сегодняшний день существует много зрелых инструментов и интегрированных сред разработки для широко используемых языков программирования. Эти инструменты развивались в течении многих лет для удовлетворения большинства потребностей индустрии разработки программного обеспечения.

Однако такие программные продукты зачастую очень сложны и имеют монолитную архитектуру, обычно автономную от инфраструктуры компилятора языка, что затрудняет замену ключевых компонентов или реализацию поддержки дополнительных языков.

В этой работе мы увидим как строилась архитектура компиляторов и какие осложнения это вызывает в современных реализациях IDE, современый подход к построению компиляторов и реализацию гибкой распределеной интегрированная среды разработки на основе концепции Language Server для мультипарадигменного языка программирования SLang[1].

Глава 1

Обзор литературы

1.1 Традиционная архитектура компиляторов

С середины 20 века исследования и индустрия разработки ПО сделали большой прорыв в разработке компиляторов с акцентом на классическую задачу компиляции: создание быстрого и эффективного объектного кода для выполнения на виртуальной машине или микропроцессоре.

Однако другие возможности компиляторов, такие как инструменты анализа кода, способные предоставлять исчерпывающую информацию о семантике программного кода, остается необычным и редким качествои современных компиляторов для популярных языков программирования.

Наиболее наглядно эту проблему можно наблюдать в инструментах разработки для Java:

Код Java обычно компилируется с использованием компилятора Sun Java. "Будучи монолитной программой, построенная по принципу “черного ящика"[2], Sun Java свособен лишь принять программный и сгенерировать из него байткод для виртуальной машины Java.

В то же время современная среда разработки включает в себя набор инструментов, упрощающих работу программиста, и требует предварительного анализа синтаксиса и семантики кода, что не представляется возможным без построения Семантического Представления[2]. Построение такого представления требует реализовать основную функциональность компилятора заного.

Глядя в прошлое, легко понять причины этой проблемы: традиционно программы рассматривались как простые текстовые объекты преобразующиеся в исполняемый код. Согласно этому предположению, компиляторы были спроектированы очень логично: они не не хранили никакой информации о семантике исходного кода, помимов некоторых низкоуровневых внутренних промежуточных представлений.

Промежуточные представления этих компиляторов имели очень ограниченный набор вариантов использования [2, 3], более того, они хорошо подходили для единственной задачи: генерации объектного кода для числа поддерживаемых микропроцессоров. Также внутреннее представление компилятора нестабильно и имеет тенденцию очень активно меняться во время разработки компилятора [4]. Следовательно, внутреннее представление компилятора не может быть использовано для построения средств анализа кода.

Ситуация с инструментарием C++ еще более удручающая: синтаксис языка и семантика намного сложнее, чем у Java, поэтому создание альтернативного компилятора — весьма непростая задача даже для большого бизнеса.

В результате мы можем наблюдаеть заметный недостаток инструментов для C++[3], а существующие довольно сложны в реализации: среда разработки JetBrains CLion имеет собственный парсер и семантический анализатор для реализации функций автодополнения, рефакторинга и статического анализа, основанные на собственном семантическом представлении языка C++. Будучи сложным программным продуктом, парсер CLion, как правило, имеет собственные недостатки и отстает от развития языка на несколько месяцев после выхода нового стандарта.

Microsoft Visual Studio "страдает"от той же проблемы: VC++ генерирует промежуточное представление, пригодное только для генерации кода: внутреннее представление сильно фрагментировано и очень низкоуровнево. Набор инструментов C&C++ IntelliSense в интегрированной среде разработки Microsoft Visual Studio полностью отделен от компилятора VC++ и реализует свой собственный парсер и семантический анализатор.

1.2 Современные компиляторы и СП

Несмотря на то, что традиционные компиляторы сегодня широко используются, их возможности по интеграции в среды разработки давно исчерпаны и всё больше новых языков сейчас нацелены на реализацию семантического представления как стабильного промежуточного представления, которое можно использовать во внешних инструментах семантического анализа.

Согласно [2, 3], в отличие от промежуточного представления традиционного компилятора, семантическое представление содержит полный спектр “знаний” о программе, и включает в себя все аспекты, которые подразумеваются в исходном коде. Это дает возможность строить мощные инструменты, основанные на семантическом анализе.

- генерация кода

- распределенная (или рекурсивная) проверка синтаксиса
- человекопонятная визуализация
- статический анализ
- интерпретация программы
- Семантический поиск: очень мощный метод запроса семантических объектов кода (пример: "найти все классы, производные от класса C, которые не переопределяют виртуальную функцию f")

Существует три основных пути описания семантического представления и предоставления доступа к нему извне: предоставление доступа к программному интерфейсу, который реализует доступ и модицикацию элементов семантического представления[4, 5] отображение семантического представления на реляционную базу данных[6], и предоставление доступа к семантическому представлению в виде текста, используя открытый формат структурирования данных[7].

Цитируя [2], "API является универсальным способом реализации любой требуемой функциональности, но в условиях меняющихся требований, невозможно предугадать спектр потребностей клиентов". Открытый формат семантического представления, в свою очередь может стать решением потенциальной проблемы: "Обычно доступ к открытому формату можно получить несколькими способами: от простых API до высокоуровневых специалилированных программных продуктов. Кроме того, возможна реализация собственных инстерфейсов обработки семантического представления, выраженного в открытом формате" And an open SP format can be a solution to potential problems: "open formats

Конкретным форматом может являться как нечто собственной разработки, так и унифицированное стандартное решение, такое как XML[8] или JSON[9].

1.3 LSP and distributed approach to building development environment

Учитывая вышеизложенное, в настоящее время мы имеем прочную основу для обеспечения хорошего инструментария, основанного на семантическом анализе: методов представления семантического представления исходного кода программного обеспечения и его оценки в соответствии с потребностями клиентов.

Современные IDE применяют эти методы для предоставления достойного сервиса, но все же существует проблема: эти программные продукты используют свои собственные реализации компиляторов, обычно несвободные и не связанные с командой разработчиков исходного языка. Это подразумевает набор проблем, отмеченных в **??**.

Наличие хорошего современного компилятора, способного генерировать SR, делает вещи немного менее сложными, но все же не решает проблему времени и стоимости реализации IDE для конкретного языка.

Очевидно, что эти проблемы не являются уникальными для класса продуктов IDE, но для любой большой монолитной архитектуры, и решение может быть довольно простым: если мы можем снизить связывание системы и представить среду разработки в виде набора инструментов вместо одного интегрированного решения, мы можем распределить реализацию IDE, чтобы иметь набор несвязанных модулей:

- Редактор кода
- Комилятор, предоставляющий семантическое представление
- Клиенты семантического представления (см. 1.1)
- Протокол, объединяющий клиенты и компилятор

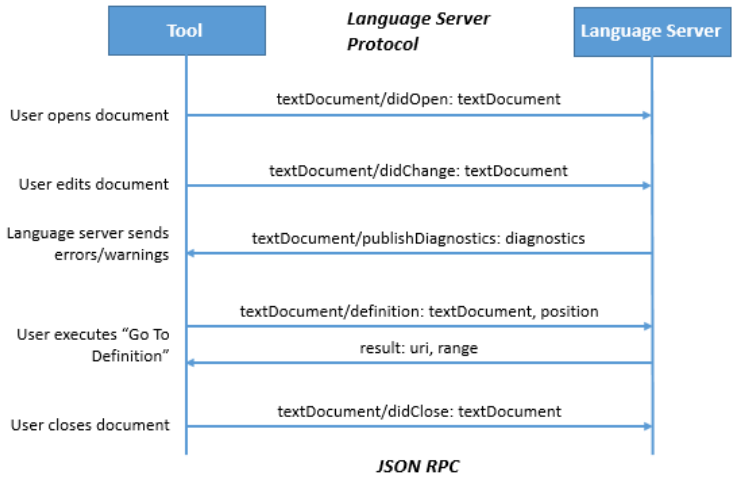

Рис. 1.1: Протокол, объединяющий клиенты и компилятор

Описанный протокол уже существует – Протокол Языкового Сервера удовлетворяет потребность в подключении сторонних инструментов, давая возможность реализовать функциональность полноценной IDE, избегая расходы на содержание и развитие отдельного монолитного программного продукта.

Языковой Сервер и ротокол языкового сервера, разработанные корпорацией Microsoft в 2016 году, предлагает представить среду разработки как две слабо связанные стороны:

- Языковой Сервер, реализующий анализ семантического представления и основанные на нём утилиты.
- Клиенты, в качестве которых выступают редаеторы кода и другие инструменты разработчика, использующие протокол языкового сервера для связи с языковым сервером[10].

1.4 Заключение

Обычные компиляторы с монолитной архитектурой, которые хороши только при генерации исполняемого кода, трудно интегрировать в современную среду разработки, поскольку они не разделяют семантическое представление исходного кода, поэтому для разработки хорошей IDE необходимо написать свой собственный исходный код компилятору семантического представления.

Современный компилятор (который предоставояет промежуточное представление высокого уровня) — это большой шаг к упрощению разработки утилит разработчика, дающий большие преимущества в затратах времени и финансов в сочетании с распределенной архитектурой IDE, разделяющей редактор и набор инструментов для анализа семантики языка на две непересекающиеся части и связывая их через стандартизированный протокол.

Такой подход дает разработчикам языка большие возможности для развития инфраструктуры языка, которые могут предоставить набор инструментов анализа через языковой сервер, а также дает возможность способ экспериментировать с новыми и существующими методами семантического анализа. Например, База Знаний о ПО[11], описанная Бертраном Мейером, может быть реализована как модуль языкового сервера, в качестве альтернативы подходу, выбранному авторами в 1985 году: интеграция инструментов анализа в редактор не представлялась возможной в то время, а это именно то для чего языковой сервер подходит наилучшим образом.

В заключение следует отметить, что языковой сервер может быть рассмотрен как наиболее приемлемое решение для быстрого развертывания богатой инфраструктуры языка, что особенно важно для новых языков программирования.

Глава 2

Методология и архитектура

2.1 Интеграция копилятора

Идея Языкового Сервера заключается в том, чтобы запускать экземпляр ЯС в том же каталоге проекта, который открыт в редакторе, и в подключении его к редактору по протоколу ЯС.

Языковой сервер отвечает за реализацию функций редактора по работе с кодом, он оперирует над семантическим представлением языка и другими метаданными, нужными для осуществения семантического анализа и, следовательно, предоставляет редактору данные в согласованном формате через протокол языкового сервера. Поскольку языковой сервер в значительной степени полагается на современный компилятор, который предоставляет SR, нам нужно реализовать способ интеграции компилятора в языковой сервер и обеспечить их взаимодействие.

Это можно сделать двумя способами: либо использовать компилятор в качестве библиотеки, либо вызывать его в отдельном процессе, передавая компилятору требуемые аргументы командной строки.2.1

вызов команды	подключение через API
более простая интеграция	сложная интеграция
ограниченное количество опций вызова	возможность выражать сложные стратегии интеграции
необходисомть в разборе формата	возможность обмениваться типами данных компилятора
необходимость реализации своего API обхода семантического представления	компилятор может экспортировать API обхода представления
необходимость описания типов данных компилятора в языковом сервере	компилятор может экспортировать внутренние типы данных

Таблица 2.1: Сравнение методик интеграции компилятора в языковой сервер

Поскольку компилятор SLang[1] не предоставляет API обхода AST или внутренних типов данных, большинство преимуществ, характерных для опции "подключение через API не применимы в нашем случае. Кроме того, компилятор предоставляет стабильное семантическое представление в формате JSON, который, будучи стандартизированным текстомым форматом, может быть легко передан через канал операционной системы, такой как standard output[12].

Таким образом, мы получаем ряд удобных свойств когда компилятор вызывается нашим языковым сервером как команда, к тому же этот вариант легче реализовать как на стороне языкового сервера, так и на стороне компилятора. И поскольку мы не ограничены какой-либо функциональностью, которая требовала бы подключения компилятора через API, мы можем считать этот способ интеграции наиболее уместным в нашем случае и придерживаться его.2.1

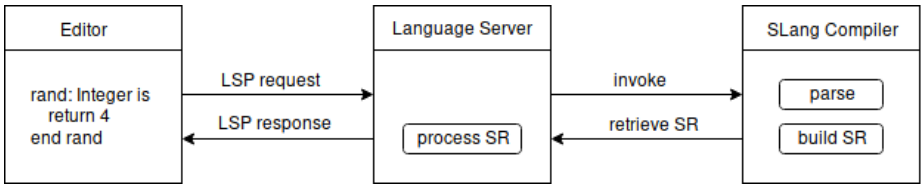

Рис. 2.1: Взаимодействие компилятора с языкомым сервером

2.2 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: 2.2

- The Core
- Module System

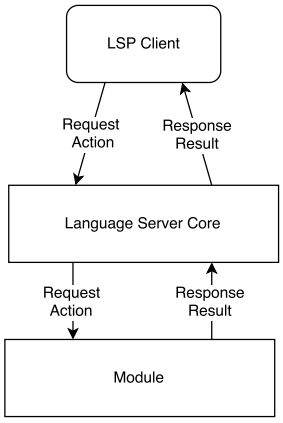

Рис. 2.2: Language Server High-level Architecture

2.2.1 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.

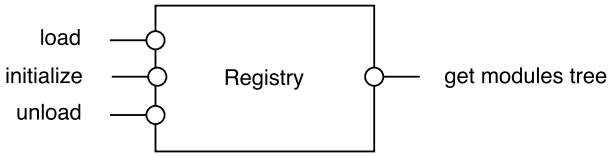

Рис. 2.3: Module Registry API

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:2.3

- 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 2.2.2.

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.

2.2.2 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.

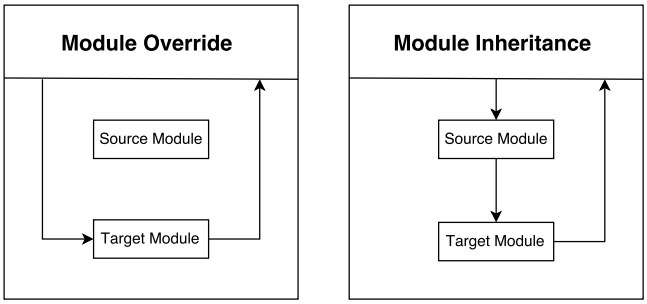

Рис. 2.4: Module Hierarchy

One of the ways to develop such architecture has already been introduced above: the hierarchy of modules expressed with the basic Object-Oriented Programming terms2.4. 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 2.5

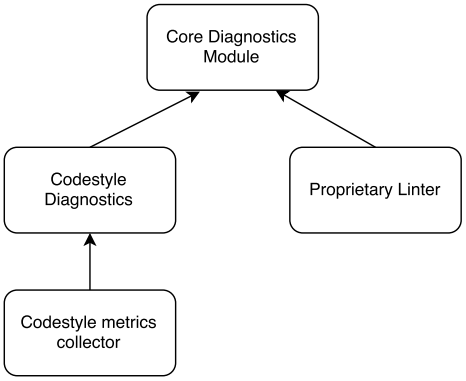

Рис. 2.5: Example Modules Tree

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 modules, 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 2.6.

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.

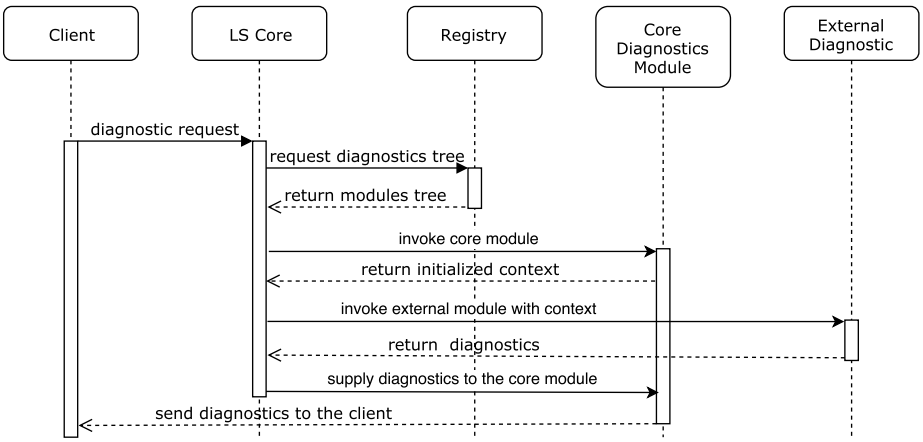

Рис. 2.6: Modules invocation sequence diagram

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.

2.3 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 2.7

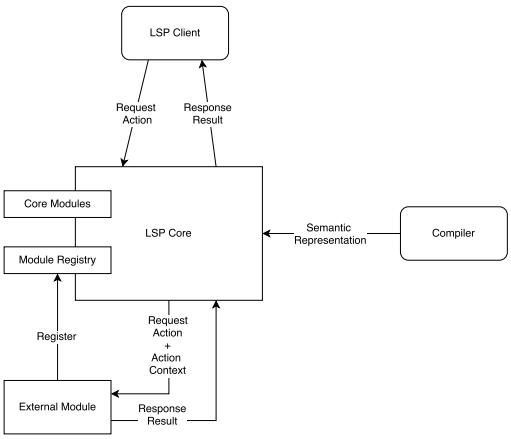

Рис. 2.7: Final architecture

2.4 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.

2.4.1 SLang Semantic Representation format

Slang has three scoping components: Unit, Block and Routine, where any nesting order is valid. Each of these components has its 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.

2.4.2 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.

2.4.3 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.

Глава 3

Evaluation and Discussion

The implementation of Akkadia is compliant with **jsonrpc** standard used in the Language Server Protocol as it inherits the implementation from Rust Language Server.

Language Server Protocol includes a list of request methods that Language Server can or should handle. Akkadia Core has an implementation to handle file notifications to update its internal Virtual File System and the mandatory service methods:

- ExitNotification
- Initialized
- DidOpen
- DidChange
- Cancel
- DidSave
- DidChangeWatchedFiles
- ShutdownRequest
- InitializeRequest

The other methods suck as the implemented **Completion** and are to be impelemted as the connectable **modules**. Module System was implemented to fully support the modular approach described in the Methodology, except the configuration files and the on-the-fly module integration.

The other Language Server methods responsible of extending code analysis capabilities of an editor are to be implemented as a further work, namely

- DidChangeConfiguration
- GoToDefinition
- ShowReferences

- Rename
- FindImpementations,
- ShowSymbols
- Formatting,
- RangeFormatting,

This can be done in any language that supports JSON encoding.

Also, continuing on module system, it may be extended by providing more auxillary data to the modules, via allowing them to have an internal storage in the **State** structure, to improve dependent (via inheritance) modules interoperation capabilities. This way Akkadia could support many languages as it was planned initially, via providing the 'Core' language modules, responsible for compiler integration, that would store the Semantic Representation in this additional storage, for the use of the modules that inherit from the core language module.

Considering the speed of execution, the current 'naive' implementation of the module IPC may be improved through the use of shared memory where it is appropriate. This way there wouldn't be performance losses on transmitting huge data structures such as the Virtual File System.

The major achievement of the Akkadia architecture and implementation is an easy to use **Module System**, that is extendable without any Rust or any other language-specific instruments, libraries, or input-output method limitations hence is providing great flexibility.

Глава 4

Conclusion

Language Server is a new way to bring modularity and extensive code reuse into modern integrated development toolsets, that employs language's compiler to perform code analysis.

With the architecture described in this thesis, this modularity may be brought to the next level, allowing to inject virtually any analysis, statistical, or other tools into the Language Server, making it possible to extend simple code editors up to a point of building very powerful toolsets. Also this approach to building a Language Server can make the concept of LS agile enough to make it an easy to use instrument for enterprise users, who are tending to have a lot of specific internal tools, which may be now incorporated into the Language Server itself.

From the perspective of a programming language developer, Language Server allows to rapidly bootstrap a ready-to-use environment of different code editors supporting the inspection tools for this new language.

The Language Server approach allows to reduce time and cost of development of the language support in major editors and IDEs at least by the cost of implementing a parser, as it is reused from the compiler. Additionaly one Language Server implementation can support a variety of different editors, while a plugin is usually able to provice a language support only for a single editor or IDE.

Concluding, the Language Server may be considered to be the most feasible solution to make a rich development infrastructure for aspiring new languages, with a broad path to evolve further through the introduced module system.

Список литературы

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

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

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

[4] Free Software Foundation. *Ada Compiler GNAT*. 2016. URL: http://www.adacore.com/home/products/gnatpro/ (дата обр. 30.09.2017).

[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 (дата обр. 30.09.2017).

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

[9] ECMA-404. “The JSON Data Interchange Format”. в: *ECMA International* 1st Editio.October (2013), с. 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/ (дата обр. 30.09.2017).

[11] Yuan Wanghong и др. “C++ program information database for analysis tools”. в: *Proceedings Technology of Object-Oriented Languages. TOOLS 27 (Cat. No.98EX224)* (), с. 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.