Санкт-Петербургский государственный университет

Кафедра системного программирования
Программная инженерия

Тюрин Алексей Валерьевич

Комплексная поддержка синтаксиса языка Vyper в IntelliJ Platform

Курсовая работа

Научный руководитель:
ст. преп. Кириленко Я. А.

Консультант:
к. ф.-м. н. Березун Д. А.

Санкт-Петербург
2019

Оглавление

Введение

Технология *блокчейн* (blockchain) постепенно находит применение во многих областях, особенно в сфере финансовых технологий. В общем случае, блокчейн — децентрализованная распределенная система для хранения и проведения транзакций в одноранговой сети. И хотя изначально блокчейн-платформы были нацелены исключительно на криптовалюты, многие из них теперь обладают поддержкой *смарт-контрактов* (smart contract). Смарт-контракты являются программируемыми объектами, хранящимися в блокчейне, и предоставляют набор функций для взаимодействия с ними извне. Такая поддержка позволяет разрабатывать произвольные приложения, часть логики которых реализуется непосредственно внутри блокчейн-платформы, что значительно расширяет область применения технологии [6].

Смарт-контракты разрабатываются с использованием специальных языков программирования и в силу специфики технологии, зачастую, управляют активами пользователей. Так как это программы, то в них возможны ошибки и потенциальные уязвимости, что приводит к потерям и хищениям активов [9]. На возникновение таких проблем влияют разные факторы: недостатки языка, расхождения между семантикой языка и интуицией программиста, особенности блокчейна как технологии, архитектура среды исполнения смарт-контрактов. В связи с этим проводится большое число исследований, результатом которых является разработка языков программирования, удовлетворяющих определенным свойствам [10].

Для успешного использования язык должен обладать определенной инфраструктурой: средой разработки, инструментами для тестирования и развертывания, различными библиотеками. Одним из последних разработанных языков является *Vyper* [8] — язык для написания смарт-контрактов для популярной блокчейн-платформы *Ethereum*. *Vyper* разрабатывается с целью улучшить безопасность смарт-контрактов, облегчить процесс их разработки, а также упростить аудит. Учитывая то, что язык эксперементальный и нахо-

дится в ранней версии, его поддержка осуществлена в малом количестве редакторов и является довольно примитивной. Принимая во внимание интерес сообщества к *Ethereum* и идеи, заложенные в основу языка, ожидается, что он станет стандартом для разработки смарт-контрактов. Таким образом, актуальной становится задача поддержки *Vyper* в одной из популярных интегрированных сред разработки (IDE), для обеспечения его необходимой инфраструктурой для ускорения и облегчения процесса разработки смарт-контрактов.

1. Постановка задачи

Целью данной работы является реализация комплексной поддержки синтаксиса для плагина языка *Vyper* в IntelliJ Platform. Так как язык находится в разработке, решение должно быть легко расширяемым и изменяемым. Для достижения цели были поставлены следующие задачи.

1. Исследовать предметную область:
    - изучить блокчейн-платформу *Ethereum*;
    - изучить язык *Vyper* и сделать обзор существующих инструментов для разработки.
2. Изучить IntelliJ Platform OpenAPI и множество инструментов для разработки плагина.
3. Реализовать комплексную поддержку синтаксиса в плагине:
    - реализовать спецификацию грамматики языка *Vyper*;
    - реализовать синтаксичесий анализатор;
    - реализовать подсветку синтаксиса;
    - реализовать навигацию по коду;
    - реализовать автодополнение;
    - реализовать проецирование сообщений от других модулей плагина в окно редактора.

2. Обзор

В этой главе рассматриваются существующие блокчейн-платформы с поддержкой смарт-контрактов и соответствующие языки программирования. Также приводится обоснование выбора платформы *Ethereum*, языка *Vyper* и описание их характеристик. В конце главы представлен обзор существующих решений, осуществляющих поддержку синтаксиса языка *Vyper*.

2.1. Смарт-контракты

Смарт-контракт — программа, хранящаяся в блокчейне и предназначенная для расширения возможностей транзакций в сети. Зачастую реализуется как набор функций, каждая из которых может служить точкой входа в смарт-контракт. Такие функции вызываются посредством транзакций и могут принимать как входные параметры, так и какое-то количество *токенов*[1], а также имеют доступ к состоянию блокчейна или смарт-контракта и могут вызывать функции в других смарт-контрактах, не создавая транзакций, т.е. атомарно.[2]

2.2. Блокчейн-платформы

Реализация смарт-контрактов для расширения возможностей транзакций в сети блокчейн была предложена вместе с *Ethereum* [2] — платформой для разработки распределенных децентрализованных приложений, которые используют блокчейн для хранения своего состояния. *Ethereum* является самой крупной блокчейн-платформой на сегодняшний день, поддерживающей смарт-контракты [1].

Помимо *Ethereum* существует множество других платформ, поддерживающих в том или ином виде смарт-контракты: Corda [5], Tezos [4], Hyperledger Fabric [3] и др.. В основном они различаются в используе-

[1]Определенное количество криптовалюты, представляющей активы пользователей.

[2]Характеристики смарт-контрактов зависят от реализации в конкретной платформе, например, некоторые платформы не имеют состояния, однако принципы для всех систем общие.

мых алгоритмах консенсуса[3], предоставляемых бизнес-ориентированных механизмах и используемых языках для смарт-контрактов.[4] *Ethereum* обладает открытым исходным кодом, постоянно развивается, хорошо документирована, имеет много инструментов и большое сообщество, поэтому в работе в качестве платформы для смарт-контрактов выбор пал именно на *Ethereum*.

2.3. Vyper

Платформа *Ethereum* имеет виртуальную машину *EVM* [12] для исполнения смарт-контрактов. Все языки, на которых ведется разработка смарт-контрактов для этой платформы, компилируются в байткод *EVM*, и контракты хранятся в блокчейне в скомпилированном виде. *EVM* имеет несколько реализаций со спецификацией ее формальной семантики и большое количество статических анализаторов, которые сообщают о потенциальных уязвимостях [7].

На данный момент основным языком для написания смарт-контрактов в *Ethereum* является *Solidity*. С этим языком связано большинство атак и уязвимостей в смарт-контрактах[5], которые вызваны плохой читаемостью, неинтуитивной семантикой и нецелостностью абстракций. Также *Solidity* является Тьюринг-полным, что несколько ограничивает процесс статического анализа.

С целью уменьшить число возможных уязвимостей в смарт-контрактах в платформе *Ethereum* был разработан язык *Vyper* [11]. Это *python*-подобный язык, который компилируются в байткод *EVM*, что позволяет переиспользовать инструменты, оперирующие напрямую с байткодом, такие как статические анализаторы или инструменты развертывания. Дизайн языка нацелен на безопасность, понятность синтаксиса и целостность абстракций. Также язык не поддерживает рекурсию, про-

[3]Алгоритмы для достижения согласованного состояния в одноранговой децентрализованной сети.

[4]Некоторые языки имеют специфицированную формальную семантику

[5]`https://www.wired.com/2016/06/50-million-hack-just-showed-dao-human/`

извольные циклы, перегрузку функций и наследование и является не Тьюринг-полным, что упрощает статический анализ. Язык обладает строгой статической типизацией и имеет встроенную проверку на переполнение для некоторых типов данных.

2.4. Обзор существующих решений

Синтаксическая поддержка языка *Vyper* существует в нескольких редакторах и некоторые из них также осуществляют поддержку различных инструментов разработки. Все известные реализации основаны на грамматике языка *Python*, что влияет на качество подсветки и автодополнения[6]. По этой же причине синтаксические ошибки подсвечиваются лишь после компиляции.

Это обусловлено тем, что язык находится в бета-версии и даже официальная реализация компилятора не имеет формально специфицированной грамматики.[7]

Плагины для *Vim*[8], *Emacs*[9], *Atom*[10] предоставляют подсветку синтаксиса, основанную на грамматике языка *Python* и регулярных выражениях.

Плагин для *VSCode*[11] обеспечивает подсветку синтаксиса, имеет предопределенные шаблоны кода, интегрированный компилятор и статический анализатор. Основан на грамматике *Python*, поэтому ошибки в коде программы показывает лишь после компиляции.

Онлайн IDE *Remix*[12] поддерживает базовую подсветку синтаксиса и не обеспечивает навигацию по коду. Также основан на грамматике языка *Python*, поэтому указывает на ошибки в коде лишь после компиляции и недостаточно информативно. IDE была разработана специально для языка *Solidity*, а так как основа у языков одна, то *Remix*

[6]Например редактор может предложить вставить метод из *Python*, которого естественно нет в *Vyper*

[7]`https://github.com/ethereum/vyper/issues/1363`

[8]**`https://github.com/jacqueswww/vim-vyper`**

[9]**`https://github.com/ralexstokes/vyper-mode`**

[10]**`https://github.com/wschwab/language-vyper`**

[11]**`https://github.com/wschwab/language-vyper`**

[12]**`https://remix.ethereum.org/`**

предоставляет большой набор инструментов для разработки: средства тестирования, развертывания и *CLI* (интерфейс командной строки) для работы с сетью *Ethereum*.

Таким образом, существующую поддержку языка можно значительно улучшить, если предъявить формальную грамматику для языка, так, чтобы синтаксическая поддержка была согласована с грамматикой, а также обеспечить интеграцию с существующими инструментами для разработки.

3. Реализация

В этой главе описываются детали реализации, архитектура и используемые технологии.

3.1. IntelliJ Platform

В качестве целевой платформы, для которой реализуется плагин, была выбрана IntelliJ Platform[13], так как она обладает следующими свойствами.

- является основой для целого набора IDE, что позволяет использовать плагин в любой из них;
- предоставляет не только SDK, но и широкий набор инструментов в виде плагинов, позволяющих облегчить и ускорить разработку плагина;
- имеет открытый исходный код платформы и большое количество референсных реализаций других плагинов с открытым исходным кодом;
- архитектура платформы позволяет вести параллельную разработку над многими компонентами системы, отдельные компоненты регистрируются в предоставляемом xml файле;

3.2. Используемые технологии

В этой секции приведена мотивация выбора того или иного инструмента разработки.

3.2.1. Язык

Разработку плагина для IntelliJ Platform можно вести только на *JVM* языках. Основным языком разработки был выбран *Kotlin*, так

[13]https://www.jetbrains.com/opensource/idea/

как он позволяет писать более компактный код по сравнению с *Java*, более безопасен, в нем больше возможности написания участков кода в функциональном стиле и более удачная организация классов по файлам, что позволяет сократить количество файлов в проекте. Также он хорошо интегрируется с *Java* и системами сборки, которые использует IntelliJ Platform.

3.2.2. Парсер-генератор

В настоящее время сложилась практика использования генераторов синтаксических анализаторов для разработки синтаксических анализаторов языков программирования, которые генерируют анализатор по входному набору правил — грамматике. Это позволяет сократить время разработки и не писать синтаксический анализатор вручную, также сгенерированные анализаторы достаточно хорошо указывают на синтаксические ошибки и легко расширяются или модифицируются, так как достаточно поправить входную грамматику.

Для генерации синтаксического анализатора используется парсер-генератор *Grammar Kit*. Для него существует поддержка в IntelliJ Platform, он позволяет вставлять в грамматику определенные пользователем функции, предоставляет механизм восстановления после ошибок, синтаксически анализирует участки, задаваемые леворекурсивными правилами грамматики.

В результате работы синтаксического анализатора получается синтаксическое дерево, состоящее из *Program Structure Interface* (PSI) элементов — абстракцией IntelliJ Platform для элементов языка, включая файлы, директорию и проект. Также он позволяет определить набор лексем для лексического анализатора и генерирует код для него, используя *JFlex*[14].

[14]Генератор лексических анализаторов: https://jflex.de/

3.3. Архитектура и элементы системы

Архитектура программного решения приведена на Рис. 1. Компонента *Editor* — часть системы, реализованная полностью в IntelliJ Platform и имеющая набор как определенных так и переопределяемых событий, обработку которых предоставляют другие компоненты, зависящие непосредственно от нее. Компонента *PsiElementsTree* включает в себя синтаксический анализатор языка, а также все внутреннее представление языка в виде классов, реализующих интерфейс *VyperPsiElement*. Программа, написанная в редакторе в определенный момент времени, синтаксически анализируется данной компонентой каждый раз, когда файл с программой изменяется, для получения абстрактного синтаксического дерева из *VyperPsiElement* объектов. Компонента *Annotators* содержит обработчики элементов *VyperPsiElement* для дополнительного синтаксического и семантического анализа дерева для подсветки ошибок, а также обработки сообщений от интегрированных инструментов. Обработчики запускаются инкрементально только на изменившихся элементах дерева. *ReferenceResolver* обеспечивает навигацию по коду, реализуя разрешение ссылок для элементов *VyperPsiElement*, а компонента *CompletionProvider* определяет набор элементов *VyperPsiElement* выше по дереву, которые могут быть подставлены на текущую позицию в редакторе.

3.3.1. Грамматика и синтаксический анализатор

Для реализации спецификации грамматики были использованы документация языка и существующий компилятор для определения корректных с его точки зрения синтаксических конструкций. Для полноценной поддержки важно, чтобы поток лексем программы разбирался до конца, т.е. чтобы синтаксический анализатор не застревал в какой-то момент. Для этого в *Grammar Kit* имеются специальные инструкции, позволяющие восстанавливаться после ошибок: синтаксический анализатор может принять правило, даже если чего-то не хватает и, наоборот, может дополнительно пропустить что-то лишнее. Также для вос-

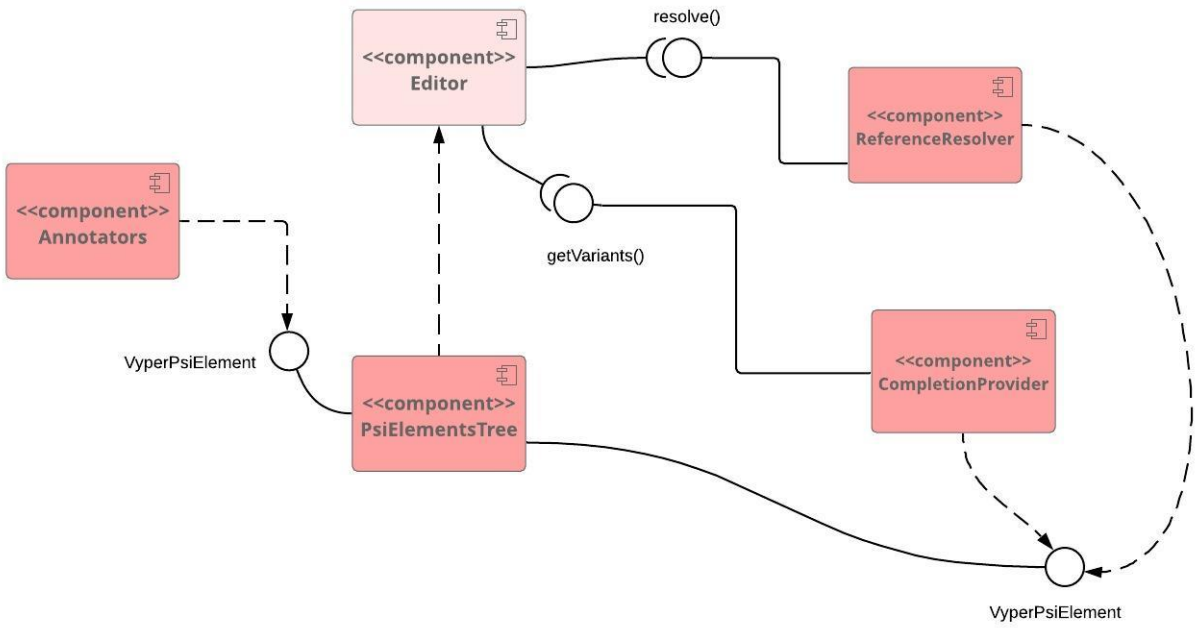

Рис. 1: Архитектура программного решения

становления от ошибок были введены специальные правила с низким приоритетом, которые подходят всегда.

Стоит отметить, что язык обладает сильной контекстной зависимостью, например, вложенность синтаксических элементов определяется уровнем отступа, а порядок объявлений в файле строго задан. Определение корректной вложенности элементов реализовано посредством функций, которые во время синтаксического анализа подсчитывают количество предшествующих пробелов у лексем и кэшируют их, что позволяет в дальнейшем использовать информацию об отступах для сообщения об ошибках, а также не пересчитывать отступы каждый раз, что влияет на производительность, т.к. синтаксический анализатор часто откатывается в силу рекурсивного спуска.

3.3.2. Подсветка синтаксиса и ошибки

Подсветка синтаксиса задается на уровне лексем и заключается в разбиении их на группы и предоставлении доступа к цвету лексемы по ее типу. Процесс подсветки ошибок начинается после построения дерева, на этом этапе можно, например, проверить разрешимость ссы-

лок для тех элементов, которые могут иметь ссылки, а также на этом уровне реализована проверка правильности порядка объявлений, чтобы излишне не усложнять грамматику. Такие обработчики синтаксического дерева запускаются инкрементально только на измененных элементах дерева.

3.3.3. Навигация

Элементы, на которые можно ссылаться должны обладать идентификатором, т.е. это различные объявления, предопределенные функции и поля. Ссылаться же может любой из следующих элементов: доступы к полям объекта, создание объекта структуры, вызовы функций и литералы. В зависимости от контекста, т.е. является ли, например, синтаксический элемент частью вызова функции или просто литералом, возвращается ссылка нужного типа. Для ссылки каждого типа определены методы в компоненте *ReferenceResolver*, которые обходят дерево в поиске объявлений с подходящим идентификатором и кэшируют результат ссылки для элемента.

3.3.4. Автодополнение

Процесс автодополнения также основан на механизме ссылок. В ответ на соответствующий запрос от пользователя IntelliJ Platform пытается подставить идентификатор, который впоследствии возвращает ссылку в зависимости от окружения. Метод *getVariants()* определяется для каждого типа ссылки и обходит синтаксическое дерево, сохраняя элементы, удовлетворяющие фильтру. В языке также присутствуют встроенные функции и поля транзакций, к которым контракт имеет доступ. Для включения их в результат поиска реализован генератор виртуальных файлов, которые содержат в себе нужные определения функций и полей. Эти файлы состоят из *PSI* элементов и в них также запускается поиск.

Заключение

В рамках курсовой работы были выполнены следующие задачи:

- изучена блокчейн-платформа *Ethereum*;
- изучен язык *Vyper* и существующие инструменты для разработки и поддержки языков смарт-контрактов;
- реализована комплексная поддержка синтаксиса языка *Vyper* для плагина в IntelliJ Platform.

Также была написана статья *"A Survey of Smart Contracts Safety and Programming Languages"*, которая была принята к выступлению на *SYRCoSE 2019* с последующей публикацией.[15]

В данный момент проводится интеграция с остальными компонентами системы, в том числе проецирование сообщений компилятора, дополнительное форматирование кода в редакторе. Стоит отметить, что имеющуюся функциональность можно значительно улучшить, покрыв больше синтаксических и семантических ошибок, расширить автодополнение и навигацию. Тем не менее плагин уже предоставляет хорошую основу и в силу использования генераторов и модульности архитектуры легко расширяем. Исходный код доступен на *GitHub*[16]

[15] `http://syrcose.ispras.ru/?q=node/12`

[16] https://github.com/NikitaMishin/vyper-plugin

Список литературы

[1] Avan-Nomayo Osato. Ethereum Surpasses Bitcoin in Number of Active Addresses. — URL: https://ethereumworldnews.com/ethereum-surpasses-bitcoin-in-number-of-active-addresses/ (online; accessed: 2018-05-31).

[2] Buterin Vitalik. Ethereum: A next-generation smart contract and decentralized application platform. — 2014. — Accessed: 2018-12-14. URL: https://github.com/ethereum/wiki/wiki/White-Paper.

[3] Elli Androulaki Artem Barger Vita Bortnikov, other. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. — 2018. — Accessed:2018-12-14. URL: https://arxiv.org/pdf/1801.10228.pdf.

[4] Goodman L.M. Tezos — a self-amending crypto-ledger White paper. — 2014. — Accessed:2018-12-14. URL: https://tezos.com/static/papers/white_paper.pdf.

[5] Hearn Mike. Corda: A distributed ledger. — 2016. — Accessed:2018-12-14. URL: https://www.corda.net/content/corda-technical-whitepaper.pdf.

[6] Macrinici Daniel, Cartofeanu Cristian, Gao Shang. Smart contract applications within blockchain technology: A systematic mapping study // Telematics and Informatics. — 2018. — Vol. 35, no. 8. — P. 2337 – 2354.

[7] Making Smart Contracts Smarter / Loi Luu, Duc-Hiep Chu, Hrishi Olickel et al. // Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security.

[8] The New Viper Smart Contract Programming Language. — URL: https://ethereumclassic.github.io/blog/2017-03-13-viper/.

[9] Nicola Atzei Massimo Bartoletti Tiziana Cimoli. A survey of attacks on Ethereum smart contracts. — 2016. — Accessed: 2018-12-14. URL: https://eprint.iacr.org/2016/1007.pdf.

[10] O'Connor Russell. Simplicity: A New Language for Blockchains. — 2017. — Accessed:2018-12-14. URL: https://arxiv.org/pdf/1711.03028.pdf.

[11] Vyper. — URL: https://github.com/ethereum/vyper.

[12] Wood Gavin. Ethereum: A secure decentralised generalised transaction ledger EIP-150 REVISION (759dccd - 2017-08-07). — 2017. — Accessed: 2018-12-14. URL: https://ethereum.github.io/yellowpaper/paper.pdf.