  
МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ  
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ  
ВЫСШЕГО ОБРАЗОВАНИЯ  
**«Дальневосточный федеральный университет»**

**ИНСТИТУТ МАТЕМАТИКИ И КОМПЬЮТЕРНЫХ ТЕХНОЛОГИЙ  
Департамент программной инженерии и искусственного интеллекта**

**РАЗРАБОТКА БИБЛИОТЕКИ PYTHON ДЛЯ КОМПИЛЯЦИИ И ОПТИМИЗАЦИИ КОДА**

КУРСОВАЯ РАБОТА  
по дисциплине «Технологии коллективной промышленной разработки информационных систем» по образовательной программе подготовки бакалавров по направлению 09.03.04 «Программная инженерия»

Выполнили:  
студенты гр. Б9120-09.03.04прогин  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Заяц А.Е.  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Матюнин Е.Я.  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Стрелов Г.С.  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Тимерханов Р.А.  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Ярош Е.Г.  
Руководитель:  
Старший преподаватель ДПИиИИ  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Иваненко Ю. С.

г. Владивосток  
2024

# Введение

Промышленная разработка информационных систем включает в себя множество этапов, начиная от разработки плана проекта, заканчивая тестированием проекта для чего, очевидно, необходимо множество специалистов различных профилей, а также унифицированные методы коммуникации между ними, с помощью которых можно разделить обязанности членов команды по их специализации.

Исходя из описанного выше необходимо использовать определенные технологии коллективной разработки для повышения эффективности работы в группе и соответствия конечного продукта заявленным требованиям.

В данной курсовой работе рассматривается задача коллективной разработки программного средства «Разработка библиотеки Python для трансляции кода в LLVM IR» и составление технической документации к данному программному средству.

Цель курсовой работы заключается в реализации библиотеки Python, которая позволит значительно ускорить работу программ за счёт трансляции кода в LLVM IR и последующей компиляции.

Для достижения поставленной цели необходимо решить следующие задачи:

* Разработка плана проекта;
* Обзор существующих решений и актуальность;
* Разработка регламента проведения инспекции;
* Разработка модели состояний задач;
* Разработка презентации проекта;
* Спецификация требований проекта;
* Разработка архитектуры проекта;
* Проектирование интерфейса;
* Разработка плана тестирования проекта;
* Разработка измерений проекта;
* Разработка перечня задач проекта;
* Разработка рекомендаций по кодированию;
* Разработка программного кода проекта;
* Составление руководства пользователя;
* Формирование матрицы трассабилити;
* Тестирование проекта;
* Выполнить необходимую работу над отчётом.

# Разработка плана проекта

План проекта — это документ, содержащий подробную информацию о проекте: исполнителях, задачах и сроках. Документ является конечным результатом этапа планирования, утверждается до начала любых работ и становится самым главным и достоверным источником информации о грядущем проекте.

Далее представлен список исполнителей проекта:

Таблица 1 – Список исполнителей проекта

|  |  |
| --- | --- |
| Роль | Участник |
| Лидер команды | Ярош Е.Г. |
| Программист 1 | Заяц А.Е. |
| Программист 2 | Матюнин Е.Я |
| Программист 3 | Стрелов Г.С. |
| Инженер сборки | Ярош Е.Г |
| Технический писатель | Тимерханов Р.А. |

Был разработан перечень задач для выполнения и примерные сроки их реализации. Описанные данные отображены на диаграмме Ганта (рисунок 1).

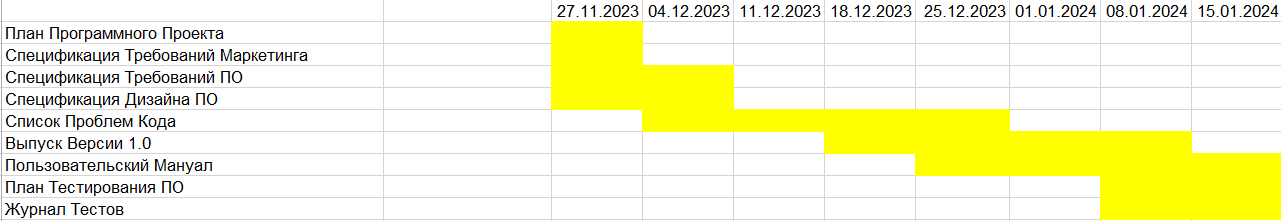


Рисунок 1. Диаграмма Ганта

# Разработка регламента проведения инспекции

Верификация рабочих продуктов является неотъемлемой частью процесса по обеспечению их качества. Современной технологией программирования выработаны специальные стандарты, подходы и механизмы проведения верификаций рабочих продуктов в формате так называемых инспекций (*peer reviews*).

Инспекция — это мероприятие по обеспечению качества рабочих продуктов проектов по разработке ПО и иной деятельности, которая проводится разработчиками, возможно – с участием представителей заказчика. Концептуально инспекция имеет следующие цели:

* Обнаружить ошибки в функциях, логике, содержании или реализации рабочих продуктов на ранних этапах их разработки и предотвратить их наследование;
* Рационально донести замысел или реализацию продукта до всех заинтересованных лиц (через их участие);
* Оптимизировать, оценить или улучшить рабочий продукт.

## Критерии формальности инспекции

Неформальная инспекция проводится:

* в случае изменения участка документа, содержащего не более 5 страниц или 10% рабочего продукта, для текстовых документов;
* в случае изменения не более 5 страниц или 10% рабочего продукта для документов дизайна;
* в случае изменения программного кода в размере не более 50 строк;

Формальная инспекция проводится в случае невозможности проведения неформальной инспекции.

## Участники инспекции

Участники могут иметь следующие роли:

* автор (author) – сотрудник, разработавший инспектируемый рабочий продукт, либо сделавший инспектируемые изменения в существующем рабочем продукте;
* председатель (moderator) – ответственный сотрудник, выполняющий роль председателя инспекции;
* инспектор (inspector) – сотрудник, ответственный за эффективную проверку инспектируемого рабочего продукта.

В инспекции в обязательном порядке присутствуют три участника, имеющие следующие роли: один автор и два инспектора, один из инспекторов может быть председателем.

## Этапы инспекции

1. Инициация — создание автором запроса на внесение изменений в продукт и назначение инспектора.
2. Подготовка и проведение — анализ изменений и внесение замечаний инспектором, если необходимо.
3. Завершение — вынесение вердикта о внесении изменений в продукт.

## Порядок организации инспекции

Работа над проектом ведётся в системе контроля версий GIT. Автор изменений оформляет Pull Request и выбирает двух инспекторов посредством рабочего чата социальной сети ВКонтакте, при этом упоминая инспекторов.

Столкнувшись с изменением дизайна проекта, инспектор обращается к председателю инспекции в рабочем чате во *ВКонтакте*. По окончании своей работы инспектор одобряет *Pull Request* или отправляет на доработку автору, уведомляя его в рабочем чате во *ВКонтакте*.

## Порядок подготовки и проведения инспекции

Инспекция должна быть проведена в течение семи дней с момента её объявления.

После анализа изменений инспектор оставляет в GIT замечания, обозначая степень их важности. При наличии серьёзных замечаний автору необходимо их исправить. Иначе инспекция считается завершённой и изменения вступают в силу.

## Перечень статусов и степени важности замечаний

1. Комментарий — замечание по оптимизации продукта, не требующее обязательных изменений.
2. Ошибка — замечание, требующее срочных исправлений.

## Порядок верификации учёта замечаний

После вынесения повторных изменений инспектор сравнивает изменения и вынесенные замечания. Если нет дополнительных замечаний, то выносится вердикт верификации и изменения вступают в силу, иначе назначаются новые замечания.

## Метрики, характеризующие эффективность инспекций

Inspection Efficiency (IE): IE = Размер продукта / (Общее время инспекции (в часах) \* Количество исправлений).

Inspection Fault Density (IFD): IFD = (Количество найденных ошибок / Размер рабочего продукта)

# Разработка модели состояний задач

Каждая задача отражает проходящий деловой процесс и проходит определенные состояния. Сначала требуется создать задачу, затем идет выполнение работ по задаче, после выполнения всех работ задача завершается.

## Перечень возможных состояний задач и их интерпретация

* *Todo* — новые задачи, которые в дальнейшем необходимо выполнить.
* *In progress* — задачи, выполнением которых уже занимается участник(-и).
* *Testing* — задачи, нуждающиеся в проверке работоспособности изменений.
* *To review* — задачи в состоянии проверки изменений на соответствие требованиям проекта.
* *Done* — выполненные задачи.

## Правила создания новой задачи

Любой участник команды разработки может в любое время создавать задачи в рамках назначенной ему части проекта.

## Правила перехода задачи из состояния в состояние

В качестве системы отслеживания задач используется *GitHub Projects*.

* При создании задача имеет состояние «*Todo*». Участник с ролью *Team Leader* может назначать на задачу исполнителя и инспектора, также участник команды может взять роль исполнителя задачи на себя.
* Как только участник команды разработки готов приступить к выполнению задачи, на которую он был назначен, задача переводится им в состояние «*In progress*». На этом этапе исполнитель выполняет задачу.
* Когда задача выполнена, участник завершивший ее переводит состояние задачи в «*Testing*». На этом этапе исполнителю необходимо проверить работоспособность изменений на наборе тестов, иначе говоря, выполнить тестирование.
* По завершении процесса тестирования, задача переходит в состояние «*To review*». На этом этапе проходит инспекция изменений. При необходимости внесения изменений состояние задачи необходимо изменить на «*In progress*», иначе – на «*Done*».

# Разработка презентации проекта

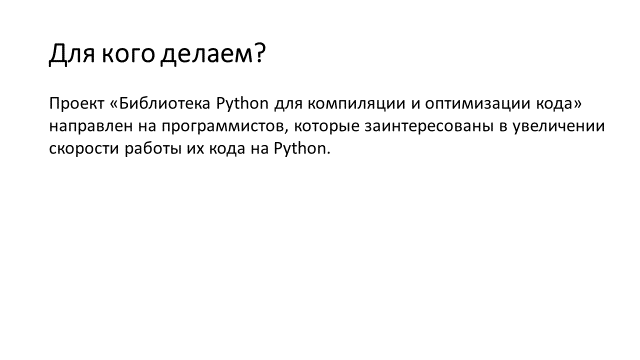
В рамках работ над проектом была разработана презентация, которая состоит из 20 слайдов. На рисунках 2 – 21 продемонстрированы слайды данной презентации.



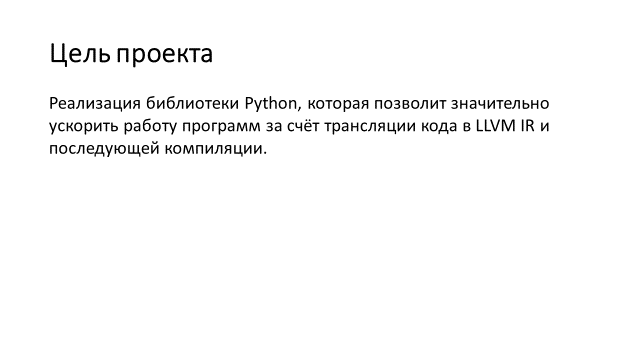
*Рисунок 2. Титульный слайд*



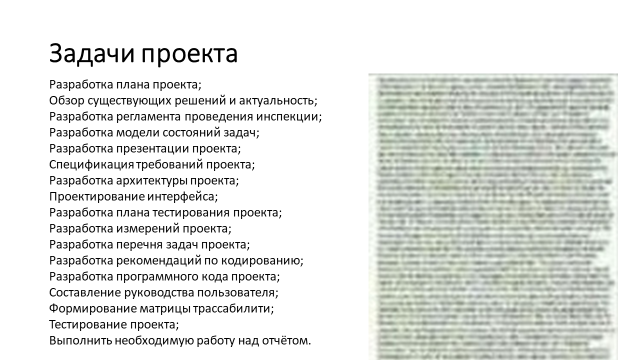
*Рисунок 3. Команда разработки*



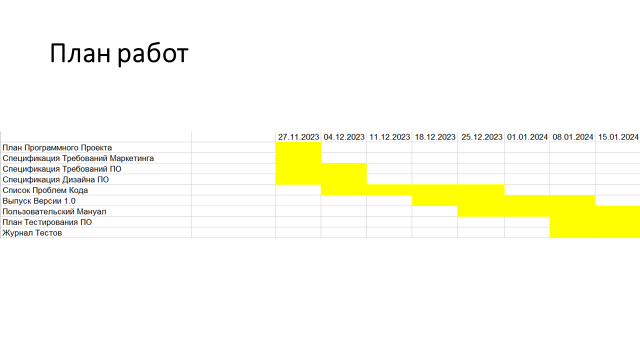
*Рисунок 4. Заинтересованные группы пользователей*



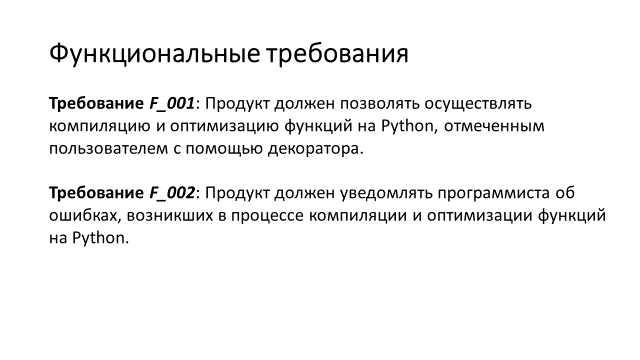
*Рисунок 5. Цель проекта*



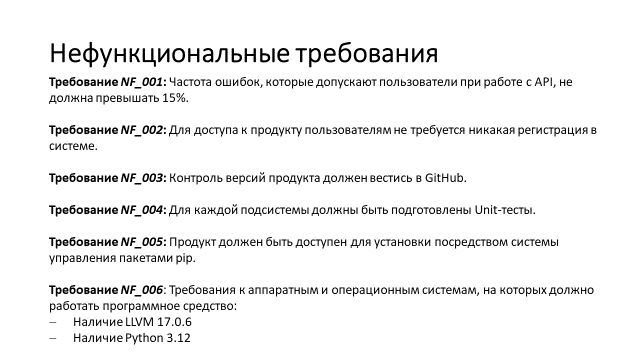
*Рисунок 6. Задачи проекта*



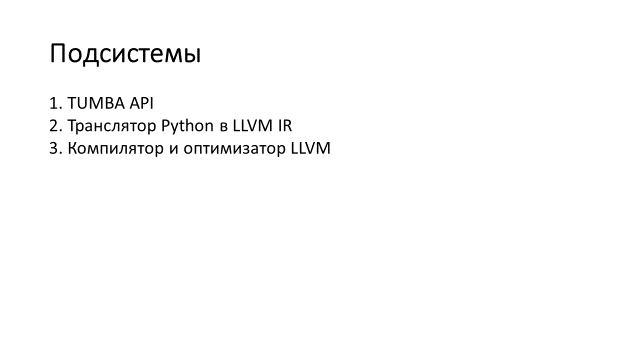
*Рисунок 7. План работ*



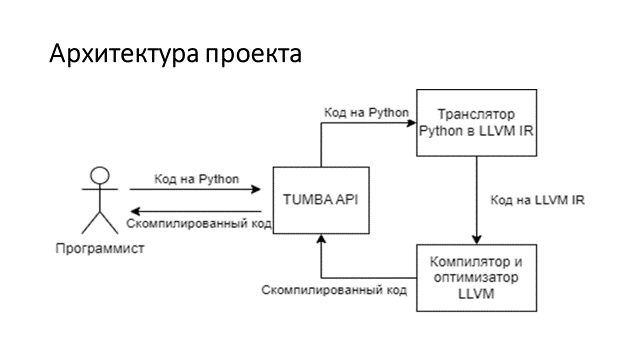
*Рисунок 8. Функциональные требования*



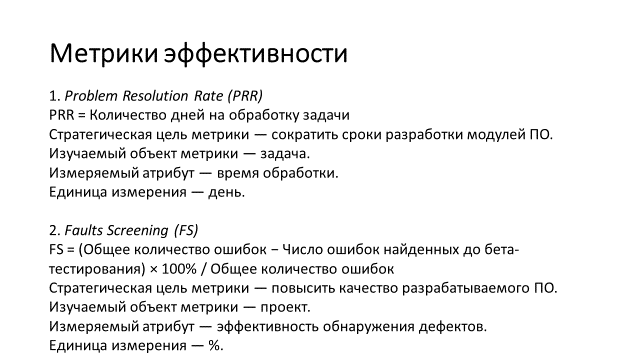
*Рисунок 9. Нефункциональные требования*



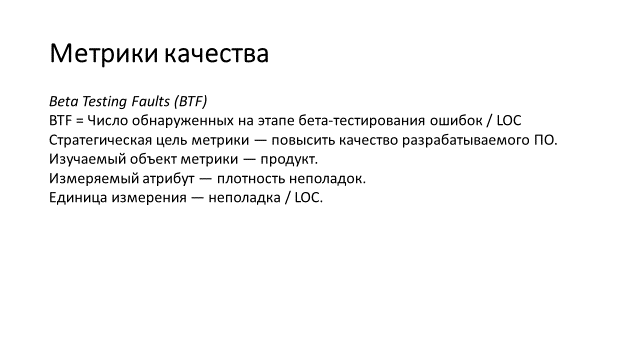
*Рисунок 10. Подсистемы проекта*



*Рисунок 11. Архитектура проекта*



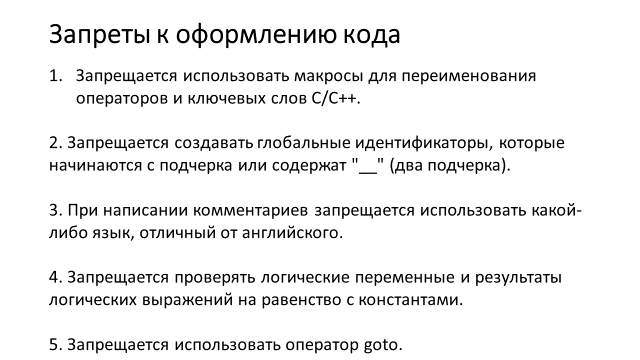
*Рисунок 12. Метрики эффективности процесса разработки*



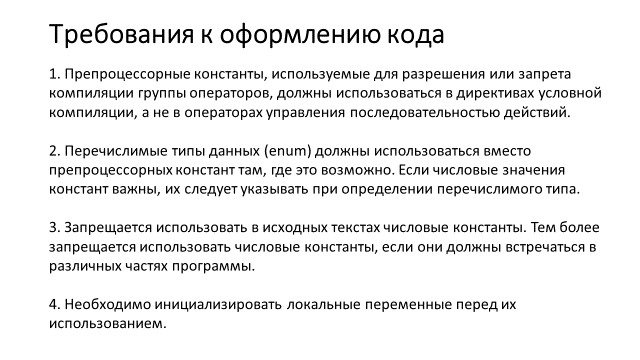
*Рисунок 13. Метрики качества программного продукта*



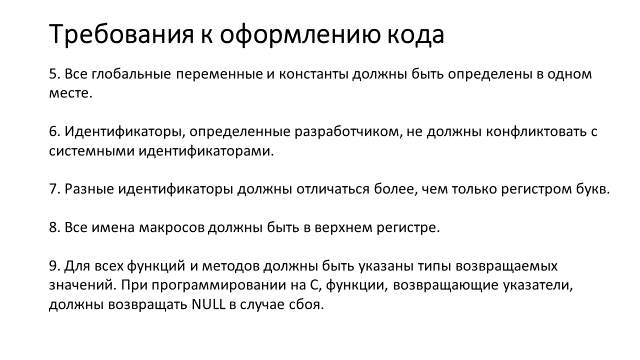
*Рисунок 14. Рекомендации по кодированию – заголовок раздела*



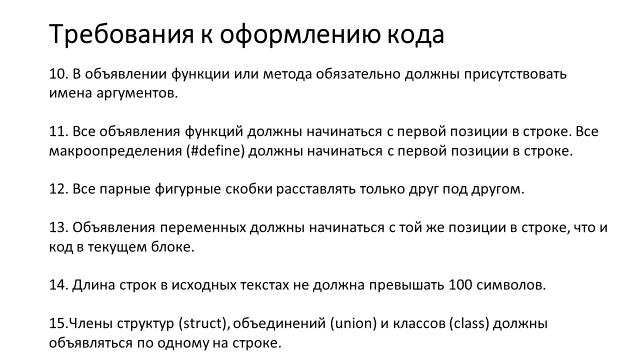
*Рисунок 15. Запреты к оформлению кода*



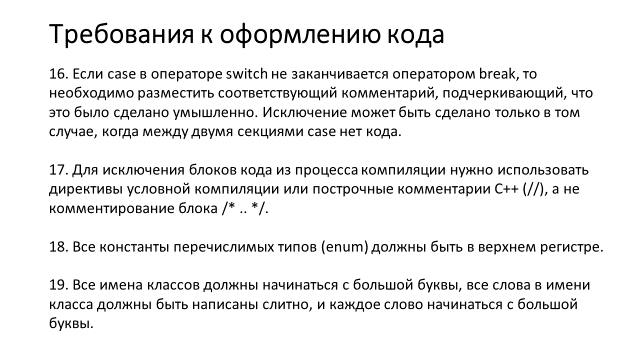
*Рисунок 16. Требования к оформлению кода 1*



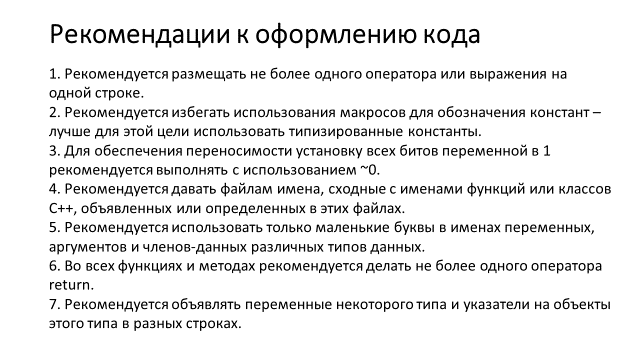
*Рисунок 17. Требования к оформлению кода 2*



*Рисунок 18. Требования к оформлению кода 3*



*Рисунок 19. Требования к оформлению кода 4*



*Рисунок 20. Рекомендации к оформлению кода*



*Рисунок 21. Спасибо за внимание!*

# Разработка требований к проекту

Программный продукт: библиотека Python для компиляции и оптимизации кода.

Программный продукт предназначен для компиляции и оптимизации кода на Python.

Программный продукт состоит из следующих подсистем:

1. TUMBA API
2. Транслятор Python в LLVM IR
3. Компилятор и оптимизатор LLVM

## Общие требования

В данном разделе описываются функциональные и нефункциональные требования к проекту.

### Функциональные требования

Требование *F\_1*: Продукт должен предоставить декоратор для обёртки функций Python.

Требование *F\_2*: Продукт должен транслировать код Python в эквивалентный код на LLVM.

Требование *F\_3*: Продукт должен позволять осуществлять оптимизацию, компиляцию и запуск эквивалентного кода на LLVM, отмеченным пользователем с помощью декоратора.

Требование *F\_4*: Продукт должен уведомлять программиста об ошибках, возникших в процессе компиляции и оптимизации функций на Python.

### Нефункциональные требования

Требование *NF\_1*: Частота ошибок, которые допускают пользователи при работе с API, не должна превышать 15%.

Требование *NF\_2*: Для доступа к продукту пользователям не требуется никакая регистрация в системе.

Требование *NF\_3*: Контроль версий продукта должен вестись в *GitHub*.

Требование *NF\_4*: Для каждой подсистемы должны быть подготовлены *Unit*-тесты.

Требование NF\_5: Продукт должен быть доступен для установки посредством системы управления пакетами pip.

Требование *NF\_6*: Требования к аппаратным и операционным системам, на которых должно работать программное средство:

* Наличие LLVM 17.0.6
* Наличие Python 3.12

## Требования к подсистеме «TUMBA API»

Требование *API\_1*: Данная подсистема должна передавать код отмеченной функции подсистеме «Транслятор Python в LLVM IR» в корректном виде.

Требование *API\_2*: Данная подсистема должна возвращать информацию о завершении работы прочих подсистем.

## Требования к подсистеме «Транслятор Python в LLVM IR»

Требование *TRANS\_1*: Подсистема должна обеспечивать корректные лексический анализ входного кода и передачу лексем синтаксическому анализатору.

Требование *TRANS\_2*: Подсистема должна обеспечивать корректные синтаксический анализ лексем и передачу синтаксического древа семантическому анализатору и генератору кода, либо передачу ошибки синтаксиса подсистеме «TUMBA API».

Требование *TRANS\_3*: Подсистема должна обеспечивать корректный семантический анализ синтаксического древа.

Требование *TRANS\_4*: Подсистема должна обеспечивать корректные генерацию кода LLVM на основе синтаксического древа и передачу кода подсистеме «Компилятор и оптимизатор LLVM».

## Требования к подсистеме «Компилятор и оптимизатор LLVM»

Требование *CUM\_1*: Подсистема должна обеспечивать корректную компиляцию кода на LLVM и все возникающие при компиляции ошибки выводить в консоль.

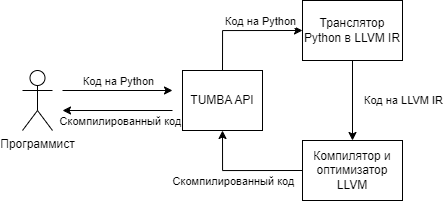
Требование *CUM\_2*: Подсистема должна обеспечивать корректную оптимизацию кода (сокращение арифметических выражений, избавление от строчек кода с неиспользуемыми данными).

# Разработка архитектуры проекта

Разработка архитектуры проекта – важный этап в рамках процесса описания и создания любого продукта, в том числе и программного. Архитектура определяет структуру проекта и взаимодействия между его компонентами.

Архитектурно-контекстная диаграмма (АКД) используется для представления взаимодействия различных подсистем проекта между собой, также она позволяет показать обмен данными между ними, что, в свою очередь,

Конкретно для разрабатываемой системы была разработана АКД, изображенная на рисунке 22.



*Рисунок 22. Архитектурно-контекстная диаграмма проекта*

С помощью декоратора, предоставляемого *TUMBA API*, программист сможет выбрать функции в коде на Python, которые необходимо компилировать и оптимизировать.

Обёрнутые декоратором функции обрабатываются транслятором Python в LLVM IR, где код функции транслируется в соответствующее представление.

Результат трансляции затем обрабатывается *компилятором и оптимизатором LLVM*, где код на LLVM IR подвергается оптимизации и компилируется.

Декоратор, предоставленный *TUMBA API*, завершая работу, уведомляет программиста об успешной или неуспешной компиляции кода.

Помимо АКД системы, была построена диаграмма потоков данных. Диаграммы потоков данных используются для представления иерархии функциональных процессов и потоков данных в программной системе. Они демонстрируют, как каждый процесс преобразует входные данные в выходные и отображают отношения между этими процессами.

Для разрабатываемой системы была разработана следующая диаграмма потоков данных (рисунок 23).



*Рисунок 23. Диаграмма потоков данных*

# Разработка измерений проекта

Мониторинг производственного процесса и его результатов является одним из важных аспектов работы при разработке программного средства. Из-за специфики программного продукта для оценки эффективности процесса и качества конечного продукта используются специальные методы. Набор действий, направленных на количественную оценку работы компании, называется программой измерений компании. Измерения проводятся как в рамках отдельных проектов, так и в отношении конкретных аспектов работы компании.

## Метрики эффективности процесса разработки

1. *Problem Resolution Rate (PRR)*

PRR = Количество дней на обработку задачи.

Стратегическая цель метрики — сократить сроки разработки модулей ПО.

Изучаемый объект метрики — задача.

Измеряемый атрибут — время обработки.

Единица измерения — день.

PRR = 150 дней / 6 этапов = 25 дней на этап.

2. *Faults Screening (FS)*

FS = (Общее количество ошибок − Число ошибок, найденных до бета-тестирования) × 100% / Общее количество ошибок

Стратегическая цель метрики — повысить качество разрабатываемого ПО.

Изучаемый объект метрики — проект.

Измеряемый атрибут — эффективность обнаружения дефектов.

Единица измерения — %.

FS = 54 ошибок всего × 100% / 54 ошибок всего = 100%

## Метрики качества программного продукта

*Beta Testing Faults (BTF)*

BTF = Число обнаруженных на этапе бета-тестирования ошибок / LOC

Стратегическая цель метрики — повысить качество разрабатываемого ПО.

Изучаемый объект метрики — продукт.

Измеряемый атрибут — плотность неполадок.

Единица измерения — неполадка / LOC.

BTF = 54 ошибки на бета-тесте / 2850 LOC = 0,001894

# Разработка перечня задач проекта

# Разработка рекомендаций по кодированию

Для разработки качественного, легко читаемого и понятного кода на любом языке программирования, важно придерживаться установленных стандартов и принципов. Это особенно важно при совместной разработке программного обеспечения. Целью любого стандарта кодирования является установление набора правил, которые способствуют более последовательному стилю кода и снижению вероятности распространенных ошибок.

## Запреты к оформлению кода

1. Запрещается использовать макросы для переименования операторов и ключевых слов С/С++.
2. Запрещается создавать глобальные идентификаторы, которые начинаются с подчерка или содержат "\_\_" (два подчерка).
3. При написании комментариев запрещается использовать какой-либо язык, отличный от английского.
4. Запрещается проверять логические переменные и результаты логических выражений на равенство с константами.
5. Запрещается использовать оператор goto.

## Требования к оформлению кода

1. Препроцессорные константы, используемые для разрешения или запрета компиляции группы операторов, должны использоваться в директивах условной компиляции, а не в операторах управления последовательностью действий.
2. Перечислимые типы данных (enum) должны использоваться вместо препроцессорных констант там, где это возможно. Если числовые значения констант важны, их следует указывать при определении перечислимого типа.
3. Запрещается использовать в исходных текстах числовые константы. Тем более запрещается использовать числовые константы, если они должны встречаться в различных частях программы.
4. Необходимо инициализировать локальные переменные перед их использованием.
5. Все глобальные переменные и константы должны быть определены в одном месте.
6. Идентификаторы, определенные разработчиком, не должны конфликтовать с системными идентификаторами.
7. Разные идентификаторы должны отличаться более, чем только регистром букв.
8. Все имена макросов должны быть в верхнем регистре.
9. Для всех функций и методов должны быть указаны типы возвращаемых значений. При программировании на C, функции, возвращающие указатели, должны возвращать NULL в случае сбоя.
10. В объявлении функции или метода обязательно должны присутствовать имена аргументов.
11. Все объявления функций должны начинаться с первой позиции в строке. Все макроопределения (#define) должны начинаться с первой позиции в строке.
12. Все парные фигурные скобки расставлять только друг под другом.
13. Объявления переменных должны начинаться с той же позиции в строке, что и код в текущем блоке.
14. Длина строк в исходных текстах не должна превышать 100 символов.
15. Члены структур (struct), объединений (union) и классов (class) должны объявляться по одному на строке.
16. Если case в операторе switch не заканчивается оператором break, то необходимо разместить соответствующий комментарий, подчеркивающий, что это было сделано умышленно. Исключение может быть сделано только в том случае, когда между двумя секциями case нет кода.
17. Для исключения блоков кода из процесса компиляции нужно использовать директивы условной компиляции или построчные комментарии C++ (//), а не комментирование блока /\* .. \*/.
18. Все константы перечислимых типов (enum) должны быть в верхнем регистре.
19. Все имена классов должны начинаться с большой буквы, все слова в имени класса должны быть написаны слитно, и каждое слово начинаться с большой буквы.

## Рекомендации к оформлению кода

1. Рекомендуется размещать не более одного оператора или выражения на одной строке.
2. Рекомендуется избегать использования макросов для обозначения констант – лучше для этой цели использовать типизированные константы.
3. Для обеспечения переносимости установку всех битов переменной в 1 рекомендуется выполнять с использованием ~0.
4. Рекомендуется давать файлам имена, сходные с именами функций или классов C++, объявленных или определенных в этих файлах.
5. Рекомендуется использовать только маленькие буквы в именах переменных, аргументов и членов-данных различных типов данных.
6. Во всех функциях и методах рекомендуется делать не более одного оператора return.
7. Рекомендуется объявлять переменные некоторого типа и указатели на объекты этого типа в разных строках.

# Разработка плана тестирования проекта

*Таблица … Матрица покрытия*

|  |  |
| --- | --- |
| Тестовые ситуации | Тестируемые требования |
| TEST\_PIP | NF\_5 |
| TEST\_API | API\_1, API\_2, NF\_1, NF\_2, NF\_6, F\_1 |
| TEST\_LEX | TRANS\_1, NF\_1, NF\_2, NF\_6, F\_1, F\_2 |
| TEST\_PARSE | TRANS\_2, NF\_1, NF\_2, NF\_6, F\_1, F\_2 |
| TEST\_SEM | TRANS\_3, NF\_1, NF\_2, NF\_6, F\_1, F\_2 |
| TEST\_GEN | TRANS\_4, NF\_1, NF\_2, NF\_6, F\_1, F\_2 |
| TEST\_CUM\_1 | CUM\_1, NF\_1, NF\_2, NF\_6, F\_1, F\_2, F\_3, F\_4 |
| TEST\_CUM\_2 | CUM\_2, NF\_1, NF\_2, NF\_6, F\_1, F\_2, F\_3, F\_4 |

*Код тестовой ситуации:* TEST\_PIP.

*Смысл ситуации.* Попытка установить продукт посредством pip.

*Ход проведения теста*. Открываем терминал, вводим «pip install tumba», ожидаем результат выполнения команды.

*Ожидаемый результат.* Продукт установлен.

*Код тестовой ситуации*: TEST\_API.

*Смысл ситуации*. Попытка получить сообщения о завершении работы подсистем и состоянии переданного транслятору кода от API.

*Ход проведения теста.* Запускаем программу с функцией, обёрнутой декоратором, в режиме отладки. Ожидаем сообщения.

*Ожидаемый результат.* Сообщения о завершении работы подсистем соответствуют действительности – сообщение об ошибке при ошибке, сообщение об успешном завершении работы при отсутствии ошибок.

*Код тестовой ситуации*: TEST\_LEX.

*Смысл ситуации*. Проверить корректность лексического анализа входного кода и передачи лексем синтаксическому анализатору.

*Ход проведения теста.* Запускаем программу с функцией, обёрнутой декоратором, в режиме отладки. Ожидаем сообщения о работе и последовательность лексем.

*Ожидаемый результат.* Сообщения о завершении работы подсистемы соответствуют действительности – сообщение об ошибке при ошибке, сообщение об успешном завершении работы при отсутствии ошибок. Последовательность лексем имеет требуемый для корректной работы и соответствующий действительности вид.

*Код тестовой ситуации*: TEST\_PARSE.

*Смысл ситуации*. Проверить корректность синтаксического анализа лексем и передачи синтаксического древа семантическому анализатору и генератору кода, либо передачи ошибки синтаксиса подсистеме «TUMBA API».

*Ход проведения теста.* Запускаем программу с функцией, обёрнутой декоратором, в режиме отладки. Ожидаем сообщения о работе и представление синтаксического древа.

*Ожидаемый результат.* Сообщения о завершении работы подсистемы соответствуют действительности – сообщение об ошибке при ошибке, сообщение об успешном завершении работы при отсутствии ошибок. Синтаксическое древо имеет требуемый для корректной работы и соответствующий действительности вид.

*Код тестовой ситуации*: TEST\_SEM.

*Смысл ситуации*. Проверка корректности семантического анализа синтаксического древа.

*Ход проведения теста.* Запускаем программу с функцией, обёрнутой декоратором, в режиме отладки. Ожидаем сообщения о работе и представление размеченного синтаксического древа.

*Ожидаемый результат.* Сообщения о завершении работы подсистемы соответствуют действительности – сообщение об ошибке при ошибке, сообщение об успешном завершении работы при отсутствии ошибок. Размеченное синтаксическое древо имеет требуемый для корректной работы и соответствующий действительности вид.

*Код тестовой ситуации*: TEST\_GEN.

*Смысл ситуации*. Проверка корректности генерации кода LLVM на основе синтаксического древа и передачи кода подсистеме «Компилятор и оптимизатор LLVM».

*Ход проведения теста.* Запускаем программу с функцией, обёрнутой декоратором, в режиме отладки. Ожидаем сообщения о работе и сгенерированный код.

*Ожидаемый результат.* Сообщения о завершении работы подсистемы соответствуют действительности – сообщение об ошибке при ошибке, сообщение об успешном завершении работы при отсутствии ошибок. Сгенерированный код на целевом языке эквивалентен исходному коду на исходном языке.

*Код тестовой ситуации*: TEST\_CUM\_1.

*Смысл ситуации*. Проверка корректности компиляции кода на LLVM и вывода всех возникающих при компиляции ошибок в консоль.

*Ход проведения теста.* Запускаем программу с функцией, обёрнутой декоратором, в режиме отладки. Ожидаем сообщения о работе и сгенерированный код.

*Ожидаемый результат.* Сообщения о завершении работы подсистемы соответствуют действительности – сообщение об ошибке при ошибке, сообщение об успешном завершении работы при отсутствии ошибок. Сгенерированный код на целевом языке эквивалентен исходному коду на исходном языке.

*Код тестовой ситуации*: TEST\_CUM\_2.

*Смысл ситуации*. Проверка корректности оптимизации кода.

*Ход проведения теста.* Запускаем программу с функцией, обёрнутой декоратором, в режиме отладки. Ожидаем сообщения о работе и сгенерированный код.

*Ожидаемый результат.* Сообщения о завершении работы подсистемы соответствуют действительности – сообщение об ошибке при ошибке, сообщение об успешном завершении работы при отсутствии ошибок. Оптимизационные изменения внесены в сгенерированный код, при этом оптимизированный сгенерированный код эквивалентен неоптимизированному сгенерированному коду.

# Тестирование проекта

# Заключение

# Список литературы

1. Гриняк В.М. Лекции по дисциплине «Технологии коллективной промышленной разработки информационных систем». Электронный вариант.