**Министерство образования и науки Российской Федерации**
**Московский физико-технический институт (государственный университет)**

Физтех-школа радиотехники и компьютерных технологий
Кафедра микропроцессорных технологий в интеллектуальных системах управления
Лаборатория (OS LAB)

Выпускная квалификационная работа магистра

# Поддержка union типов в статическом языке программирования

**Автор:**
Студент M01-206 группы
Акмаев Алексей Михайлович

**Научный руководитель:**
Добров Андрей Дмитриевич

**Научный консультант:**
Бронников Георгий Кириллович


Москва 2024

**Аннотация**

Поддержка union типов в статическом языке программирования
*Акмаев Алексей Михайлович*

В данной работе исследуются способы поддержки union типов в статическом языке программирования, а также варианты генерации байткода для union типов и его оптимизации.

Работа включает: todo

Ключевые слова: union типы; компилятор; MyTS; байткод; нормализация;

Цель работы: реализация union типов в статическом TS-подобном языке программирования с дальнейшей оптимизацией байткода.

Задачи:

- Реализация базовых union типов
- Нормализация union типов
- Поддержка доступа к полям
- Внедрение литералов в union типы
- Написание lowering фаз для корректной кодогенерации
- Оптимизация байткода

**Abstract**

Support for union types in a static programming language

# Содержание

# 1 Введение

## 1.1 Описание проблемы

Известно, что разработка хороших, многоразово используемых библиотек является очень сложной задачей. В популярных статических объектно-ориентированных языках, таких как Java и C++, наследование и подтипизация (а в последнее время и обобщения в том числе) используются в качестве основных механизмов, способствующих многоразовому использованию кода. В то время как наследование позволяет одному классу повторно использовать реализацию другого класса, например, объявления переменных и сигнатур методов, подтипирование предназначено для взаимозаменяемости. Под взаимозаменяемостью подразумевается такое свойство сущности, что если объект одного типа может быть использован в определенном месте, то и другой объект, являющийся его подтипом, также может быть использован в том же месте. Взаимозаменяемость может быть также перефразирована как возможность повторного использования контекстов в том смысле, что если некоторый контекст применим к объекту одного типа, то тот же контекст также применим к любому объекту его подтипа. Для ясности дадим определения подтипа и супертипа. Подтип - это тип, являющийся производным от другого типа, который называют супертипом. Подтип наследует свойства и поведение своего супертипа, но также может добавлять дополнительные свойства или переопределять существующие. Подтипирование - это способ выразить, что один тип является специализированной версией другого. Супертип предоставляет общее определение, которое может быть расширено или специализировано с помощью его подтипов. Таким образом, проблемы проектирования, связанные с отношениями наследования и подтипирования, несколько различаются: для наследования необходимо учитывать, как новые классы могут повторно использовать существующую реализацию, а для подтипирования - как объекты могут использоваться в клиентском коде.

В популярных языках связь между подтипами по большей части основана на отношениях наследования. Исключением являются только wildcards в Java 5.0. Может случиться так, что два класса, используемые в схожих контекстах, но с довольно разными реализациями, будут разделены в иерархии классов наследования, что приведет к отсутствию полезного супертипа этих классов. Интерфейсы, как программная конструкция, в Java являются решением этой проблемы: можно определить суперинтерфейс классов схожего назначения, независимо от заданной иерархии наследования, и пользоваться преимуществами подтипирования. Однако интерфейсы не могут быть добавлены после определения класса, поэтому разработчикам библиотек по-прежнему приходится много работать над планированием иерархий интерфейсов перед выпуском библиотеки в релиз. Эта проблема считается существенным ограничением систем типов, основанными на взаимозаменяемости, как в Java.

### 1.2 Предлагаемое решение

В этой работе предлагается решение — объединение или union типы. Union типы или объединения - это тип данных в некоторых языках программирования, позволяющий переменной хранить значение, которое может быть одним из нескольких различных, в том числе несвязанных между собой наследованием, но фиксированных типов. Только один из типов, входящих в объединение, может быть ассоциирован с переменной в рантайме в конкретный момент времени. Они позволят решить проблему невозможности добавления супертипов к существующим типам, таким как классы и интерфейсы.

На практике различают два вида объединения - тегированный и нетегированный union. Нетегированное объединение можно представить как фрагмент памяти, который используется для хранения переменных разных типов данных. Как только ему присваивается новое значение, существующие данные перезаписываются новыми данными. Область памяти, в которой хранится значение, не имеет внутреннего типа (кроме просто байтов или слов памяти). Однако это значение можно рассматривать как один из нескольких абстрактных типов данных, имеющий тип значения, которое было последним записано в область памяти. Нетегированные объединения обычно довольно ограничены в использовании и представлены только в не типобезопасных языках программировании, таких как C. Тегированное объединение можно рассматривать как тип с несколькими компонентами, каждая из которых должна быть корректно обработана при манипулировании этим типом. Говоря об объединении, по умолчанию будет подразумеваться тегированный union.

Поскольку объединения состоят из существующих типов, они дают возможность определять супертип даже после того, как иерархия классов установлена. Как следует из названия, тип объединения обозначает объединение множества заданных типов, рассматриваемых как наборы экземпляров, которые принадлежат к этим типам, и на уровне рантайма ведут себя как наименьший супертип. Объединения могут использоваться не только как механизм полиморфизма в сигнатурах функций и при объявлении переменных, но для прямого доступа к полям объектов при соблюдении некоторых условий, как обычные типы. Фактически, для некоторых типов их объединяющий тип можно рассматривать как интерфейс, который "выделяет" их общие черты, то есть поля с одинаковыми именами и методы с похожими сигнатурами. Предполагается, что объединения могут быть полезны для группировки независимо объявленных классов со схожими интерфейсами, но несвязанными друг с другом никакими отношениями, а также для реализации гетерогенных коллекций, таких как списки, в которых строки и целые числа одновременно могут являться их элементами.

### 1.3 Фундамент для построения и внедрения решения

Фундаментом для исследования послужит проект по разработке некоторого статического объектно-ориентированного языка программирования с TS-подобным синтаксисом. Назовем этот язык MyTS и впоследствии будем использовать для него это название. В этом проекте также уже присутствует спецификация этого языка, которая в основном и будет определять поведение объединений. Кроме того, нам даны виртуальная машина, на которой будет исполняться исходный код, а также набор инструкций байткода (ISA). Упрощенно продемонстрируем схему исполнения программы в данном проекте.


Рис. 1: Общая схема исполнения программы

Основная логика будет реализована во фронтенде - часть проекта, осуществляющая компиляцию, семантический анализ и кодогенерацию из исходного кода на MyTS в байткод, который впоследствии исполняется на виртуальной машине.

# 2 Постановка задачи

## 2.1 Проблематика

Как уже было сказано ранее, разработка качественных и многократно используемых библиотек это довольно сложная задача. Объединения или union типы в статически типизированных языках решают эту и несколько других значимых проблем. Позволяя переменной хранить значение, которое может быть одним из нескольких указанных типов, объединения повышают безопасность типов, гибкость и выразительность кода, сохраняя при этом преимущества статической проверки типов. Кроме того, перед нашим статическим языком стоит проблема максимальной совместимости с TypeScript. Поскольку union типы часто используются в TS, например в nullish типах или опциональных полях классов и параметрах функций, необходимо реализовать ту же функциональность и в MyTS.

## 2.2 Цель работы

Поддержать union типы на уровне фронтенда, при этом соответствуя синтаксису и семантике TypeScript.

## 2.3 Задачи

- Спроектировать и реализовать класс для базовых union типов (let x: A|B = new A())
- Поддержать доступ к общим полям всех составляющих
- Реализовать нормализацию типов, входящих в объединение
- Перенести необходимый функционал в lowering фазу
- Поддержать литералы в качестве составляющих объединения
- Оптимизировать AST дерево с целью уменьшения байткода или ускорения рантайма

## 2.4 Обработка результатов

- Обеспечить достаточное покрытие тестами
- Убедиться, что процент пройденных тестов больше 80
- Привести наглядные графики результатов оптимизации

## 2.5 Требования

- Соответствовать спецификации языка MyTS
- Генерировать валидный байткод
- Не допускать просадки перформанса во время исполнения

# 3 Обзор существующих решений

Здесь надо рассмотреть все существующие решения поставленной задачи, но не просто пересказать, в чем там дело, а оценить степень их соответствия тем ограничениям, которые были сформулированы в постановке задачи.

# 4 Исследование и построение решения задачи

Для того, чтобы внедрить наше решение в существующий проект по разработке компилятора MyTS, сначала необходимо тщательно изучить его внутреннее устройство. Составим обзор компонент фронтенда для понимания, как сделать наше решение качественной и логичной частью всего проекта.

## 4.1 Исследование внутреннего устройства фронтенда

### 4.1.1 Лексер

Этот компонент преобразует исходный код в последовательность токенов. Входные данные должны быть корректной строкой UTF8. Токены могут быть литералами, знаками препинания или ключевыми словами, представленными свойством `Token::type`. Поскольку JS содержит контекстуальные ключевые слова, например, `static` - это ключевое слово внутри тела класса, но в других местах это простой идентификатор, токены имеют дополнительное поле `Token::keywordType`, которое всегда соответствует соответствующему ключевому слову независимо от реального `Token::type`. Поскольку на этом уровне могут возникать синтаксические ошибки, лексер может выдавать соответствующую ошибку.

### 4.1.2 Область видимости

Структуры `binder::Scope` - это конструкции, в которых хранятся переменные. Каждая область видимости имеет родителя - область по вложенности выше, все объявления `binder::Decl`, которые хранятся в таблице переменных `Scope::bindings`. Эта таблица содержит строку в качестве ключа и переменную `binder::Variable` в качестве значения.

### 4.1.3 Объявления

Объявления типа `var a` или `let b` во время синтаксического анализа преобразуются в `binder::Decl`. Каждое объявление знает имя и AST-узел, с которым оно связано.

### 4.1.4 Переменные

Переменные по умолчанию не создаются. Декларации преобразуются в переменные, если они проходят проверку в рамках области видимости. Структура переменной `binder::Variable` содержит в себе объявление, из которого она взята, а также имеет `checker::Type`, который будет представлять фактический статический тип переменной. Этот тип неизвестен во время синтаксического анализа и заполняется позже компонентом `Checker`.

### 4.1.5 Binder

Этот компонент создает и проверяет все привязки(bindings) деклараций к области видимости. Сам по себе он не является отдельным анализом. Каждая проверка привязки запускается в процессе синтаксического анализа. В настоящее время триггерами могут быть:

- Создание новой области видимости.
- Добавление объявления в текущую область видимости. Если она не может добавить привязку, возникает синтаксическая ошибка.

### 4.1.6 Парсер

Парсер является одним из основных компонент и взаимодействует с binder-ом и лексером одновременно. Синтаксический анализ является однопоточным, и входные данные обрабатываются только один раз. Для парсинга выбран синтаксический анализатор LR(1), поэтому он видит только следующий токен, но может заглянуть в следующую точку кода. Однако из-за новых возможностей стандарта ES2015 список параметров лямбда функций и шаблоны деструктурирования больше не могут быть корректно прочитаны, заглядывая только на один токен вперед. В этих сценариях синтаксический анализатор работает в отказоустойчивом режиме, что означает, что он следует менее строгой грамматике, чем стандартная. Всякий раз, когда закрывающий токен для этих языковых элементов найден или не найден, выполняется обход построенного AST дерева и проверка его правильности в соответствии с правилами грамматики. По мере того как AST строится во время синтаксического анализа, также создается дерево областей видимости. Как только парсер обрабатывает очередной блок кода или функцию, запускается binder, который создает для них новую область видимости и привязывает их к ней. Поскольку время жизни этих областей совпадает со временем жизни соответствующих узлов AST дерева, структуры представляющие узел также сохраняют у себя эти области видимости. Всякий раз, когда считано объявление переменной, запускается binder для проверки привязки. Таким образом, в общем случае синтаксический анализатор уведомляет binder только о том, что он обнаружил новое начало области видимости или объявление новой переменной.

### 4.1.7 AST дерево

ASTNode - это базовый класс всех узлов, сгенерированных парсером. Узел AST хранит в себе данные о позиции в исходном коде, родительский узел и другие атрибуты, которые добавляются дочерним классам при наследовании.

### 4.1.8 Анализ имен переменных

После того, как исходный код обработан парсером, AST проходит анализ имен переменных. После него создается класс программы, содержащий AST дерево, позиции переменных в исходном коде и некоторые метаданные. Главной целью этого анализа является определение типа переменной для обеспечения эффективной многопоточной компиляции без блокировки, не считая планирования потоков. Переменная может быть локальная или лексическая. Во время обхода AST дерева мы ссылаемся на уже сгенерированные области видимости, присвоенные statement узлам, чтобы определить, в какой области мы на самом деле находимся. Всякий раз, когда мы оказываемся в узле идентификатора переменной ir::Identifier, анализатор пытается разрешить ее тип из текущей области. Если у переменной нет конфликтов с какой-либо другой переменной из области видимости binder::VariableScope (чаще всего это binder::FunctionScope, иногда binder::LoopScope), переменная объявляется как локальная. В противном случае она получает лексический индекс из ближайшей области видимости и помечается как лексическая. Каждый раз, когда из области видимости запрашивается лексический слот на переменную, область становится лексической. Это означает, что во время компиляции в начале функции должно быть создано так называемое лексическое окружение. В этом анализе мы также определяем, является ли локальная переменная внутри объявления цикла частью замыкания внутри его тела. Результат этого анализа определяет, должны ли цикл или его декларация быть лексическими или нет. Таким образом, перед переходом непосредственно на стадию кодогенерации AST обрабатывается только один раз. Каждая область видимости содержит в себе информацию о том, нуждается ли она в лексическом окружении или нет, и каждая переменная знает, является ли она лексической или нет.

### 4.1.9 Чекер

Этот компонент семантически анализирует код, используя AST дерево, области видимости и переменные. Чекер обходит AST дерево и проверяет каждый узел, используя виртуальную функцию Checker, перегруженную для всех узлов по-своему. Когда чекер обнаруживает узел с объявлением какой-то переменной, он выполняет ее поиск во всех областях видимости по степени их вложенности друг в друга. Как только найдено объявление этой переменной в какой-то области видимости, чекер присваивает тип выражения к узлу дерева, если он задан явно, либо использует для этого вывод типов.

- При незаданной аннотации, тип объявленной переменной выводится с помощью инициализатора.
- При объявлении функции чекер создает для нее сигнатуру, которая состоит из параметров в заданном порядке и типа возвращаемого

значения. Оно в свою очередь выводится из явно указанной аннотации типа или выражения, следующего за return в теле функции.

- При объявлении интерфейса чекер создает объектный тип, который хранит в себе все поля и их типы, а также сигнатуры методов и конструкторов интерфейса.
- При объявлении псевдонима типа чекер использует тип, который этому псевдониму и присваивается.

#### 4.1.10 TsType

Как только тип объявления вычислен, ему присваивается значение ts-типа объявленной переменной. Используя структуру checker::Type, чекер может проверять на валидность выражения присваивания, бинарные или унарные операции, вызовы функций и конструкторов, доступы к полям класса и наследование. Важно отметить, что statement не создает тип, это делают только выражения.

- Statement проверяется только семантически. Например, если statement проверяется на наличие у него выражения-условия, которое вычисляется в тип void, то чекер сообщает об ошибке, поскольку выражения типа void не могут быть проверены на истинность.
- Выражения, с другой стороны, по своей сути порождают типы. Например, бинарное выражение 5 + 6 порождает числовой тип. Вот почему функция ASTNode::Check узла statement всегда возвращает значение nullptr, а функция ASTNode::Check узла выражения всегда возвращает тип, созданный этим выражением.

#### 4.1.11 Отношения

В определенный момент во время семантического анализа чекер должен соотнести различные типы друг с другом. Например, если переменной присвоено значение a = 15, чекер сравнивает тип переменной a с типом инициализатора, который представляет собой числовое литеральное выражение, приводимое к числовому типу. Если a был объявлен как let a: string, то это присвоение приведет к ошибке, поскольку тип string не может быть присвоен типу number. В зависимости от операции, используемой с переменными, существует 3 различных отношения типов:

- Отношение идентичности: отношение является истинным, если два сравниваемых типа абсолютно идентичны. Оно используется при повторном объявлении переменной или поля, и это наиболее сильное отношение.
- Отношение присваивания: отношение является истинным, если тип с правой стороны может быть присваивоен типу с левой стороны. Оно

используется при обработке присваиваний, операндов бинарных выражений, наследования (отношение базового и дочернего класса), а также для проверки совместимости типа возвращаемых выражений из return с типом возвращаемого значения, объявленного в функциях.

- Отношение приводимости: отношение является истинным, если тип в правой части может быть приведен к типу в левой части. Оно используется при работе с операторами сравнения, приведения типов и в не сильно отличается от отношения присваивания. Это самое слабое отношение.

#### 4.1.12 Сигнатуры функций

Сигнатуры создаются для функций и методов, и они могут быть явно объявлены в теле интерфейса или класса, используя следующий синтаксис: (a: number, b: string): number. Существует два типа сигнатур: сигнатуры функций и конструкторов. Сигнатуры функций используются для проверки правильности вызовов функций. Например, если объявление функции с именем func1 было объявлено с сигнатурой (a: string, b: string), то оно не может быть вызвано с меньшим или большим количеством параметров, чем 2, и аргументы вызова функции должны быть присваиваемы типу параметра сигнатуры в правильном порядке. Таким образом, чекер выдаст ошибку в любом из этих случаев: func1(1), func1(1, 2, 3), func1("foo"), func1(2, "bar"). Сигнатуры конструкторов идентичны по своим правилам. Разница лишь в том, что они используются при создании объектов и, соответсвенно, в выражения с ключевым словом new.

#### 4.1.13 Lowering фазы

Lowering преобразование - это фаза трансформаций, работающая после чекера и перед кодогенерацией, во время которой происходит обход AST дерева, преобразуя некоторые его узлы и заменяя отфильтрованные выражения более простыми и низкоуровневыми конструкциями. Стоит уточнить, что не все фазы lowering-а проходят после чекера. На самом деле, некоторые проходы трансформируют AST дерево и до семантического анализа. У каждого lowering прохода имеются предусловие и постусловие. Как видно из названий, предусловие является триггером для запуска трансформации дерева для конкретного узла, а постусловие проверяет, что преобразование было успешным и не нарушило структуру и инварианты AST дерева. Преимущества внедрения lowering проходов перед кодогенерацией следующие:

- Абстрагирует код – различные lowering фазы могут быть написаны независимо друг от друга
- Упрощает кодогенерацию

- Легкость отладки. Дает возможность распечатать и проанализировать состояние AST дерева до и после какой-то определенной трасформации.

Недостатки lowering проходов:

- Увеличение времени компиляции
- Возможная утеря отладочной информации
- Основательный семантический анализ проводится до большинства lowering проходов. После трансформаций не исключено, что состояние AST дерева станет некорректным.
- Подходит только для некоторых задач

Примеры lowering фаз, которые осуществляют трансформацию определенных конструкций языка и AST дерева:

- Обработка лямбда-функций, генерация для них объекта.
- Раскрытие объектных литералов
- Боксинг и анбоксинг переменных
- Преобразование выражений в вызовы рантайм функций
- Оптимизации

#### 4.1.14 Кодогенерация из промежуточного представления

Кодогенерация осуществляется параллельно для каждого AST узла. Класс `Gen` контролирует все функции, которые генерируют байткод или управляют ресурсами.

#### 4.1.15 Аллокация регистров

Формат исполняемого файла требует, чтобы все параметры размещались в конце локальных регистров. Поскольку в данном языке есть инициализация полей и переменных по умолчанию, а также rest параметры, от локальных регистров требуется выполнение определенных стандартных действий. Для этого нам нужно загружать соответствующие параметры в регистры. Номер соответствующего регистра зависит от количества используемых локальных регистров, что является циклической зависимостью. Для разрешения этой проблемы, была представлена следующая структура регистра:

| локальный n | локальный 1 | локальный 0 | параметр 0 | параметр 1 | параметр n |
|---|---|---|---|---|---|
| ... | 65534 | 65535 | 65536 | 65537 | ... |

Таким образом, во время генерации кода выделяемые регистры располагаются в порядке убывания, и при необходимости все spill-fill инструкции генерируются сразу в нужном месте. Как только вся кодогенерация завершена, становится известно количество всех выделенных локальных регистров, а аргументы генерируемых инструкций преобразуются следующим образом:

- Локальные регистры отображаются в диапазоне uint16_t
- Параметры вычисляются как UINT16_MAX - общее количество регистров.

#### 4.1.16 Выделение регистров для локальных переменных при кодогенерации

Каждый раз, когда очередной шаг кодогенерации попадает в область видимости блока или функции, класс Gen обрабатывает локальные переменные. На этом этапе известно, является ли переменная лексической или локальной. Лексические переменные не затрагиваются, поскольку они уже получили свой лексический индекс во время анализа имен переменных. Поскольку индекс регистра, в котором лежит локальная переменная, считывается или записывается только классом Gen, присваивание регистра на этом этапе безопасно. Всякий раз, когда область видимости заканчивается, эти регистры освобождаются и могут быть повторно использованы позже.

#### 4.1.17 Разрешение имен переменных

При генерации байткода всякий раз, когда требуется разрешить имя переменной ir::Identifier, мы используем тот же метод, что и при анализе имен переменных: начинаем поиск имени переменной из текущей области и отслеживаем, сколько областей видимости мы проходим. В зависимости от результатов поиска принимается решение, какой байткод нам нужно сгенерировать для разрешения:

- Местных
- Лексических
- Модульных
- Глобальных

переменных. На данный момент информация о типе переменной, вычисленной чекером, уже доступна, но в данный момент не используется.

#### 4.1.18 Узел промежуточного представления

Узел промежуточного представления имеет класс IRNode. Каждая инструкция из архитектуры набора команд (ISA), сгенерированная из файла isa.yml, имеет свой класс, который содержит только необходимые операнды. Исключением является ассемблерный класс Ins, который содержит список операндов каждого типа. Этими типами операндов могут быть:

- Виртуальный регистр - VReg
- Иммидиат - Imm
- Строка - StringView
- Метка - Label

Кроме того, их каждый узел промежуточного представления содержит в себе узел AST дерева, чтобы получать информацию о позиции какой-либо сущности в исходном коде для генерации отладочной информации. В конце каждой кодогенерации узлы IRNode будут преобразованы в ассемблерные инструкции Ins. Во время преобразования:

- Сохраняются строковые операнды, которые позже будут добавлены в таблицу строк сгенерированной программы
- Операнды VReg переназначаются на их реальные регистровые индексы.

#### 4.1.19 Эмиттер

Этот компонент преобразует каждый шаг кодогенерации из класса Gen в ассемблерный класс Function. Список сущностей, которые преобразуются в ассемблер:

- Метки
- Try-catch блоки
- Инструкции
- Отладочная информация

Также сохраняются все общеиспользуемые данные в класс ProgramElement, находящийся во фронтенде. Такими данными могут быть, например, строки и литералы, которые позже записываются в ассемблерный класс Program. Всякий раз, когда генерируется класс Gen, эмиттер объединяет все элементы программы промежуточного представления ProgramElement и заполняет таблицу функций уже созданными ассемблерными классами Function.

### 4.2 Схематичное устройство компилятора

Кратко проанализировав и описав выше основные компоненты фронтенда, которые позволяют трансформировать исходный код на языке MyTS в ассемблерное представление `Program` с дальнейшей оптимизацией и генерацией байткода, стоит представить обобщенную схему работы данного компилятора:

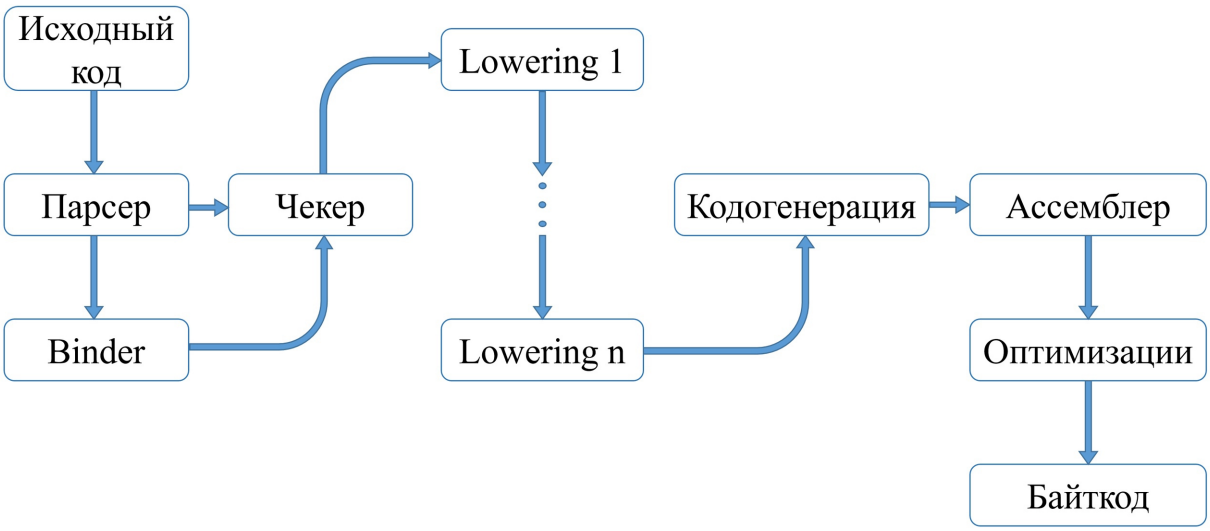


Рис. 2: Схема работы компилятора

todo: описать боксинг анбоксинг

# 5 Описание практической части

Если в рамках работы писался какой-то код, здесь должно быть его описание: выбранный язык и библиотеки и мотивы выбора, архитектура, схема функционирования, теоретическая сложность алгоритма, характеристики функционирования (скорость/память).

# 6 Заключение

Здесь надо перечислить все результаты, полученные в ходе работы. Из текста должно быть понятно, в какой мере решена поставленная задача.

Типы объединений решают несколько ключевых задач в статически типизированных языках, предоставляя гибкость в представлении данных при сохранении безопасности типов. Они улучшают возможность работы с различными формами данных, повышают ясность кода и обеспечивают более надежный и безошибочный код благодаря проверкам на этапе компиляции. Эти преимущества делают типы объединений мощной функцией для разработчиков, работающих в статически типизированных языках.

## Список литературы

[1] *Mott-Smith, H.* The theory of collectors in gaseous discharges / H. Mott-Smith, I. Langmuir // *Phys. Rev.* — 1926. — Vol. 28.

[2] *Морз, Р.* Бесстолкновительный PIC-метод / Р. Морз // Вычислительные методы в физике плазмы / Ed. by Б. Олдера, С. Фернбаха, М. Ротенберга. — М.: Мир, 1974.

[3] *Киселёв, А. А.* Численное моделирование захвата ионов бесстолкновительной плазмы электрическим полем поглощающей сферы / А. А. Киселёв, Долгоносов М. С., Красовский В. Л. // Девятая ежегодная конференция «Физика плазмы в Солнечной системе». — 2014.

## Приложение

Здесь необходимо написать приложение, которое вы должны придумать самостоятельно.