

**ВСЕРОССИЙСКИЙ КОНКУРС НАУЧНО-ИССЛЕДОВАТЕЛЬСКИХ,
ИЗОБРЕТАТЕЛЬСКИХ И ТВОРЧЕСКИХ РАБОТ ОБУЧАЮЩИХСЯ
«НАУКА, ТВОРЧЕСТВО, ДУХОВНОСТЬ»**

Направление: Информационные технологии

Тема: Проект LANGX: Мост между простотой и производительностью

Соискатель: Ош В. Э., Шелухин Д. С.

Научный руководитель: Шипшина О.В.

**Место выполнения работы: ГБПОУ РО «ТМехК», Ростовская область,
г. Таганрог.**

Содержание

	Введение	3
1	Эволюция языков программирования	4
1.1	Начало	4
1.2	Языки высокого уровня	5
1.3	Объектно-ориентированное программирование	5
1.4	Эра интернета и скриптовых языков	6
1.5	Мульти-парадигменные языки	7
1.6	Хронология языков программирования	8
2	Проект LANGX	9
2.1	Философия и цели проекта LANGX	9
2.2	Архитектура и реализация LANGX	10
2.3	Общая схема компилятора	13
2.4	Этап 1: Лексический анализ	13
2.5	Этап 2: Синтаксический анализ и построение AST	14
3	Синтаксис LANGX	19
3.1	Базовые конструкции и вывод типов	19
3.2	Функции и модульность	19
3.3	Управление памятью и безопасность	20
4	План разработки и текущий статус	22
4.1	Что уже реализовано (Прототип, v0.1.0)	22
4.2	Ближайшие цели (v0.2.0 - v0.5.0)	22
4.3	Долгосрочная перспектива	22
	Заключение	23
	Список использованных ресурсов	24

Введение

Идея создания своего языка программирования долгое время оставалась уделом крупных корпораций и академических институтов. Однако развитие мощных и безопасных системных языков, таких как Rust, наряду с доступностью образовательных ресурсов, демократизировало этот процесс. Сегодня создание компилятора — это уникальная возможность для глубокого понимания информатики, а также вызов, позволяющий воплотить собственное видение того, каким должен быть инструмент для разработчика.

LANGX — это студенческий исследовательский проект, ставящий перед собой цель: создать язык, который сочетает читаемость и простоту обучения, характерные для высокоуровневых языков (например, Python), с производительностью и безопасностью памяти, присущим современным системным языкам (таким как Rust). Мы верим, что такой «мост» может быть полезен в образовательных целях, для написания инструментов, где важна скорость, и как платформа для экспериментов.

Вдохновением для LANGX послужили не только технологические потребности, но и история IT-индустрии, полная примеров, когда смелые личные проекты и концепции, казавшиеся узконаправленными, меняли всю отрасль. Этот проект — наша попытка внести свой вклад в эту традицию, прикоснувшись к фундаментальным аспектам создания инструментов, которыми ежедневно пользуются миллионы.

1. Эволюция языков программирования

1.1. Начало

Мир программирования претерпел значительные изменения с момента своего зарождения. За эти годы было создано и продолжает развиваться множество языков, каждый из которых обладает уникальным синтаксисом, функциями и приложениями. Их важность в современном мире заключается в способности создавать различные приложения, упрощающие жизнь. Развитие языков программирования было необычайным и сыграло решающую роль в технологических достижениях.

Чарльз Бэббидж в 1830-х годах задумывает «аналитическую машину», её способность изменять поведение путем замены перфокарт с обновленными инструкциями была новаторской функцией.

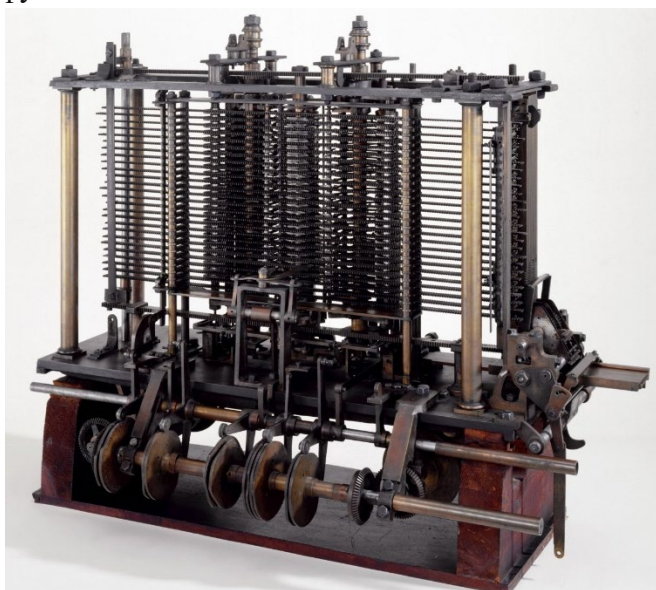


Рис. 1. Так выглядела бы машина Бэббиджа, если бы автор её всё-таки создал.



Рис. 2. Ada Lovelace

В 1843 Ада Лавлейс, которая является ученым-математиком, пишет первый алгоритм для машины Бэббиджа. Машина должна была работать на перфокартах, как ткацкий станок Жаккарда. Через много лет алгоритм Лавлейс признали первой в мире программой. Так родилась идея программирования, где самой ранней формой стал машинный код, использующий двоичные цифры (0 и 1).

Однако программирование на машинном коде является довольно-таки сложной задачей, приходилось оперировать бесконечными цепочками единиц и нулей. Написание и уж тем более проверка такого кода достаточно трудозатратны, не говоря о понимании если код был написан другим. Именно поэтому, для упрощения был создан ассемблер, суть команды в котором обозначалась сокращенными словами и буквами.

Команда под руководством Тома Килберна и Фредерика Уильямса разработали первый пример ассемблера, который использовался для Manchester Mark 1. Далее ассемблер развивался вместе с компьютерным оборудованием: добавление новых функций инструкции, полностью отвечали развитию аппаратных возможностей. Необходимо отметить, что ассемблер используется и сегодня, правда в очень узких областях. В современном программировании ассемблер чаще

всего используется для прямого управления оборудованием, доступа к специализированным инструкциям процессора или для решения критических проблем производительности.

1.2. Языки высокого уровня

С развитием технологий функционал ассемблера перестал удовлетворять потребности. Необходимо было выходить на новый уровень программирования. И один из первых языков, который появился под давлением времени, стал — Fortran. Возможность создания в текстовом виде с описанием логики выполнения с использованием циклов, ветвлений, подпрограмм, оперирование массивами и представление данных в виде действительных, целых и комплексных чисел увлекли инженеров и ученых. А благодаря своим библиотекам и научным «каркасам» Fortran до сих пор актуален, пусть и в научной среде.

Еще одна ветвь развития — С. Если Fortran стал инструментом для ученых, то С был создан для помощи программистам. Многие придерживаются мнения, что С — это продвинутый ассемблер. Несмотря на это, кодирование на С остается востребовано, чему потворствует, в первую очередь, развитие микроэлектроники: так как контроллерам, гаджетам, сетевым устройствам необходимы драйвера и другое, относительно низкоуровневое, ПО для взаимодействия с оборудованием.

1.3. Объектно-ориентированное программирование

Это высокий уровень кодирования, который все больше отходит от машинного программирования и реализуют различные парадигмы, помимо процедурных. К ним относится и реализация ООП (объектно-ориентированных принципов программирования). Java, C++, Python, Rub, JavaScript — этот спектр языков является наиболее популярным и востребованным на сегодняшний день.

ООП характеризуется предопределенными модульными единицами программирования (объектами, классами, подклассами и т. д.), предназначенными для ускорения процесса программирования и упрощения обслуживания ПО.

Алан Кей, ученый и создатель концепции ООП, выделил следующие мотиваторы развития: поиск превосходных модульных структур для сложных систем, которые включали бы сокрытие деталей и желание разработать более гибкие версии назначений. Язык программирования является объектно-ориентированным, если он поддерживает абстракцию данных и классов, наследование и полиморфизм.

Simula 1 и Simula 67, разработанные учеными из Норвегии, Оле-Йоханом Далем и Кристен Ньюгор, признаны первыми языками ООП. На ранних этапах разработки Simula ее создатели представляли себе модель системы имитации, состоящую из различных станций, каждая со своими очередями клиентов. Клиенты были доступны на всех станциях, что означало, что каждая станция могла «заимствовать» пользователя из очереди, изменять его переменные, а затем перенаправлять его в очередь другой станции. Станции могли автономно создавать или удалять клиентов и управлялись самой программой. Эта итерация, иногда называемая Simula 0, так и не достигла стадии реализации, но заложила основу для будущих объектно-ориентированных языков.

Сама идея объектно-ориентированного программирования набрала обороты в 1970-х годах, в это время, в исследовательском центре Xerox Palo Alto (PARC), был создан — Smalltalk. Отличительной чертой Smalltalk является его чистая объектная ориентация: все в коде является объектом, включая экземпляры классов, простые типы данных и даже блоки кода (замыкания).

В начале 1980-х годов Бьорн Страуструп интегрировал идею ООП в С. Получившийся язык был назван «С с классами», который в 1983 году претерпел ребрендинг и стал носить знакомое название — С++.

От визионерских идей Кристен Ньюгаард и Оле-Йохана Даля с Simula до новаторской работы Адель Голдберг и Алана Кея с Smalltalk и последующей эволюции С++ Бьорна Страуструпом были заложены основы для парадигмы, которая произвела революцию в наших представлениях о структурировании и организации кода.

Способность объектно-ориентированного программирования инкапсулировать данные и поведение в объекты, способствовать повторному использованию кода посредством наследования и полиморфизма, обеспечивать модульный подход к проектированию программного обеспечения — оказались полезными функциями в решении сложных задач современного программирования. Принципы и концепции, рожденные из этой богатой истории, продолжают определять разработку передовых приложений, фреймворков и систем.

1.4. Эра интернета и скриптовых языков

Считается, что скриптинг — это современное течение, однако и у него имеется богатая история. Скриптинг существует с тех пор, как появились компьютеры. Фактически, он был единственным способом использования компьютера в ранние дни.

В 1950-х и 60-х годах программисты отправляли перфокарты операторам мэйнфреймов и машины работали в пакетном режиме. Язык управления заданиями, JCL от ИВМ, часто упоминается как один из первых языков сценариев. Но хотя скриптовые языки были функциональными, время отклика было далеко не таким быстрым, как у современных компьютеров — часто требовалось не менее дня, чтобы получить результаты.

В 1960-х годах начали разрабатываться интерактивные системы разделения времени, и идея скриптовых оболочек вошла в практику. Одним из самых ранних проектов был MULTICS. Когда несколько программистов Bell Labs покинув проект, решили реализовать собственную систему, которую назвали UNIX. Одним из нововведений в оболочках Unix стала возможность отправлять вывод одной программы на вход другой, что позволило выполнять трудоемкие задачи в одной строке кода оболочки. В мире Unix появились и другие скриптовые языки, такие как AWK и Sed, для работы с текстом.

Другой важный язык сценариев — Perl, был изобретен в 1987-м году Ларри Уоллом, стал популярным во время бума всемирной паутины в 90-х годах при создании веб-приложений. Далее уже последовали и другие языки, такие как Python и Ruby.

Скриптовый язык — это разновидность программирования, предназначенная для облегчения создания сценариев, автоматизации задач или выполнения сложных операций. Языки сценариев часто используются для администрирования софта и веб-разработок.

Некоторые примеры популярных языков сценариев включают JavaScript, Python, PHP, Ruby. Эти языки обычно интерпретируются, то есть они выполняются строка за строкой программой, называемой интерпретатором, а не компилируются в машинный код. Это позволяет сократить время разработки и упростить отладку.

Скриптовые языки зачастую более гибкие и простые в изучении, чем традиционные языки программирования, что делает их популярным выбором как для новичков, так и для профессионалов.

В веб-разработке существует два различных подхода к выполнению скриптов: клиентские и серверные скрипты.

- Клиентские, включают запуск скриптов на компьютере или устройстве пользователя через браузер. Этот метод обычно используется для повышения интерактивности веб-сайта, например, путем проверки вводимых данных в формах или отображения динамического контента. JavaScript — наиболее широко используемый язык для клиентских скриптов.

- Скрипты на стороне сервера подразумевают выполнение сценариев на веб-сервере перед доставкой страницы в браузер пользователя. Этот метод используется для создания динамического контента на сервере, например, для доступа к базе данных или обработки данных форм.

Также языки сценариев могут использоваться в системном администрировании. Примерами скриптов языка сценариев, используемых в системном администрировании, являются Shell, Perl и Python.

1.5. Мульти-парадигменные языки

Языки программирования часто классифицируются в соответствии с их парадигмами, например: императивные, функциональные, логические, основанные на ограничениях, объектно-ориентированные или аспектно-ориентированные. Парадигма характеризует стиль, аспекты и методы языка для описания ситуаций и процессов, а также для решения проблем.

Каждая парадигма лучше всего подходит для программирования в определенных прикладных областях. Однако реальные проблемы часто лучше всего реализуются путем объединения концепций из разных парадигм, поскольку они включают аспекты из нескольких областей, и это объединение более удобно реализуется с использованием мульти-парадигменных языков программирования.

Идея мульти-парадигменного языка заключается в предоставлении структуры, в которой программисты могут работать в различных стилях, свободно смешивая конструкции из разных парадигм. Цель разработки таких языков — позволить программистам использовать лучший инструмент для работы, признавая, что ни одна парадигма не решает все проблемы самым простым, продуктивным способом.

Сама концепция мульти-парадигменного кодирования возникла с развитием методологий и языков, а ее принятие принесло огромную пользу индустрии по разработке софта, способствуя стимулированию инноваций.

Примерами языков программирования с несколькими парадигмами являются Python, JavaScript и C++. Эти языки предлагают обширные возможности для кодирования в разных стилях, что делает их очень популярными среди разработчиков. Кроме того, доступны инструменты и библиотеки, облегчающие реализацию каждой парадигмы, что ускоряет процесс разработки и оптимизирует качество получаемого кода.

Каждые несколько лет появляются новые языки программирования, которые обещают произвести революцию в разработке программного обеспечения.

1.6. Хронология создания языков программирования

1. В 1843 г Ада Лавлейс написала первый алгоритм программы.
2. Конрад Цузе создал Plankalkul в 1940-х годах. Он содержал множество последовательностей кодирования, которые инженеры и сейчас обычно используют для выполнения основных действий.
3. 1949 году создан ассемблер, предшественник современного программирования.
4. В 1952 году Autocode стал первым компилируемым языком программирования.
5. В 1957 году Джон Бэкус создал FORmula TRANslation (FORTRAN).
6. В 1958 году были изобретены ALGOL и LISP. ALGOL.
7. В 1959 году Грейс Мюррей Хоппер разработала COBOL
8. В 1964 году был создан BASIC.
9. В 1970 году Никлаус Вирт реализовал PASCAL.
10. Smalltalk, SQL и C появились в 1972 году.
11. Жан Ичбиа инициировал разработку Ады в начале 1980-х годов.
12. В 1983 Бьорн Страуструп создал C++; а Том Лав и Брэд Кокс — Objective-C.
13. В 1987 году Ларри Уолл разработал Perl.
14. Haskell был создан в 1990 году и назван в честь выдающегося математика Хаскелла Брукса Карри.
15. Visual Basic и Python появились в 1991 году.
16. В 1993 году Юкихиро Мацумото создал Ruby.
17. Java, JavaScript и PHP были впервые представлены в 1995 году.
18. Компилятор C# был создан в 2000 году.
19. В 2003 году была создана Scala, в этом же году появился Groovy.
20. Google представил свой Go в 2009.
21. В 2011 году появился Kotlin, совместимый с Java, для разработки приложений на Android.
22. В 2012 год появляются Julia, TypeScript, Elixir.
23. Apple разработала Swift в 2014 году.
24. В 2015–2016 годы появляются: Rust, Raku, Ring и Zig.

2. Проект LANGX

2.1 Философия и цели проекта LANGX

Основная философия LANGX:

1. **Доступность синтаксиса:** Код на LANGX должен легко читаться и писаться. Мы стремимся к минимализму, устранив избыточные ключевые слова и сложные конструкции там, где это возможно без потери ясности.

2. **Предсказуемая производительность:** Язык является компилируемым и статически типизированным. Он не использует виртуальную машину или сборку мусора в рантайме в классическом понимании, а полагается на семантику владения и заимствования (inspired by Rust), что позволяет генерировать эффективный машинный код и избегать накладных расходов.

3. **Образовательная ценность:** Процесс разработки LANGX — это открытый учебный проект. Мы документируем проблемы и их решения, чтобы путь создания компилятора был понятен другим студентам.

Таким образом, LANGX позиционируется не как прямая замена Rust или C++, а как исследовательская площадка и потенциальный инструмент для конкретных сценариев, где важен баланс между лёгкостью освоения и контролем над системой.

Целью создания нашего языка программирования послужила история финского программиста Линуса Торвальдса, человека, который на собственном примере показал, как увлечение, начавшееся в раннем возрасте, может перерасти в проект мирового масштаба. С детства он проявлял глубокий интерес к устройству компьютеров и принципам программирования, стремясь не просто использовать существующие технологии, а понимать их изнутри и совершенствовать. Для него программирование было не набором абстрактных команд, а инструментом самовыражения и способом влиять на окружающий мир.

Особое впечатление на нас произвело то, что Торвальдс начинал свои проекты не с целью коммерческой выгоды или признания, а из искреннего желания решить конкретную проблему и сделать что-то полезное для себя и других. Его стремление создать удобную, эффективную и открытую систему стало примером того, как личная инициатива и ответственность за результат могут привести к появлению технологий, которые объединяют миллионы людей по всему миру.

Именно эта философия легла в основу разработки нашего языка программирования. Мы вдохновлялись идеей, что язык должен рождаться из реальных потребностей разработчиков, быть понятным, честным и доступным для изучения. Как и в истории Линуса Торвальдса, ключевую роль для нас сыграло желание внести собственный вклад в развитие индустрии, предложить новый взгляд на привычные подходы и создать инструмент, который будет стимулировать обучение, эксперименты и совместное творчество.

Наш язык программирования задумывался как отражение ценностей, которые мы считаем фундаментальными: открытость, простота, практичность и уважение к сообществу. Мы верим, что даже небольшой проект, начатый из личного интереса и стремления к развитию, способен со временем перерасти во что-то значимое. История Линуса Торвальдса стала для нас напоминанием о том, что изменения в мире технологий часто начинаются с одного человека, одной идеи и искреннего желания сделать мир лучше с помощью кода.



Рис. 3. Линус Бенедикт Торвальдс — финно-американский программист, создатель ядра операционной системы Linux



Рис. 4. Логотип Linux

2.2 Архитектура и реализация LANGX

Rust был выбран в качестве языка реализации компилятора LANGX по нескольким причинам:

1. **Безопасность и производительность:** Rust гарантирует безопасность памяти без сборщика мусора, что критически важно для такого сложного проекта, как компилятор.
2. **Богатая экосистема:** Наличие отличных крейтов (библиотек) для парсинга (например, `logos` для лексического анализа, `lalrpop` или `pest` для синтаксического) значительно ускоряет разработку.
3. **Целевая платформа:** Генерация Rust-кода в качестве промежуточного представления позволяет делегировать сложнейшие задачи оптимизации и генерации машинного кода компилятору Rust, который является эталоном в этой области.

Благодаря концепциям, заимствованным у Rust, мы получили значительные преимущества в архитектуре и реализации нашего языка программирования, которые позволили создать безопасный, эффективный и гибкий инструмент для разработчиков.

Во-первых, строгая система владения памятью и заимствований, характерная для Rust, позволила нам выстраивать архитектуру языка так, чтобы исключить множество классических ошибок на уровне компиляции. Это дало возможность реализовывать сложные структуры данных и алгоритмы без риска утечек памяти или конфликтов доступа. В практическом плане это означает, что мы можем проектировать многопоточные и высоконагруженные системы, не прибегая к дорогостоящему управлению памятью во время выполнения и без необходимости постоянного контроля со стороны разработчика.

```
println!("[DEBUG] Completed binary expression: {:?}", left);
Ok(left)
}

fn parse_primary_expr(&mut self) -> Result<Expr, ParseError> {
println!("[DEBUG] Entering parse_primary_expr with token: {:?}", self.current);
match &self.current {
Token::Print => {
self.advance().map_err(|e| self.lex_err(e))?;
if self.current != Token::LParen {
println!("[DEBUG] Error: Expected '(', found {:?}", self.current);
return Err(self.err("Ожидалась ( после print"));
}
self.advance().map_err(|e| self.lex_err(e))?;
let inner = self.parse_expr()?;
if self.current != Token::RParen {
println!("[DEBUG] Error: Expected ')', found {:?}", self.current);
return Err(self.err("Ожидалась ) после выражения"));
}
self.advance().map_err(|e| self.lex_err(e))?;
Ok(Expr::Print(Box::new(inner)))
}
Token::Int(n) => {
let val = *n;
self.advance().map_err(|e| self.lex_err(e))?;
Ok(Expr::Int(val))
}
Token::Ident(name) => {
let ident = name.clone();
self.advance().map_err(|e| self.lex_err(e))?;
if self.current == Token::LParen {
```

Рис. 5. Фрагмент внутреннего кода LANGX (BACKEND)

Во-вторых, отказ от сборщика мусора и упор на статическую безопасность позволили нам создавать архитектурные решения, которые ориентированы на производительность. Например, эффективное использование стековой и кучевой памяти, строгие правила времени жизни объектов и контроль владения ресурсами дают возможность строить высокопроизводительные модули с предсказуемым временем выполнения. Это особенно важно для системного программирования, игр, серверных приложений и других сценариев, где критична каждая миллисекунда и каждый байт памяти.

В-третьих, принципы Rust позволили нам внедрить в архитектуру языка понятие безопасного и явного управления ресурсами. Это значит, что каждый объект, каждый модуль и каждая функция имеют чётко определённые зоны ответственности, что упрощает сопровождение кода и его расширение. Благодаря этому мы можем проектировать крупные проекты с минимальным количеством скрытых ошибок, при этом сохраняя модульность и гибкость системы.

```

1 <!DOCTYPE html>
2 <html lang="ru">
3 <head>
4   <meta charset="utf-8" />
5   <meta name="viewport" content="width=device-width,initial-scale=1" />
6   <title>LangX IDE - Toxic Glow</title>
7   <link rel="stylesheet" href="styles.css" />
8 </head>
9 <body>
10 <!-- WELCOME SCREEN -->
11 <div id="welcomeScreen" class="screen">
12   <canvas id="symbolCanvas" class="canvas-bg" aria-hidden="true"></canvas>
13   <div class="welcome-card">
14     
15     <h1>LANGX IDE</h1>
16     <p class="tagline">Хакни систему. Создавай. Защищай.</p>
17     <div class="welcome-actions">
18       <button id="startBtn" class="btn primary">Запустить среду</button>
19       <button id="demoAchBtn" class="btn">Тест ачивок</button>
20     </div>
21   </div>
22 </div>
23
24 <!-- APP SCREEN -->
25 <div id="appScreen" class="screen" style="display:none;">
26   <header>
27     <div class="left">
28       
29       <div class="brand"> ⚡ LangX IDE</div>
30     </div>
31     <nav class="tabs" role="tablist" aria-label="Навигация">
32       <button class="tab-btn active" data-tab="editorTab">Редактор</button>
33       <button class="tab-btn" data-tab="learnTab">Обучение</button>
34       <button class="tab-btn" data-tab="aboutTab">Об авторе</button>
35       <button class="tab-btn" data-tab="achievementsTab">Ачивки</button>

```

Рис. 6. Фрагмент кода LANGX (FRONTEND)

Кроме того, использование концепции «безопасность по умолчанию» открыло новые возможности в реализации высокоуровневых абстракций. Мы смогли создавать удобные и выразительные конструкции языка, которые позволяют разработчикам работать с памятью и ресурсами без постоянного беспокойства о безопасности. При этом при необходимости можно явно указать небезопасные операции, что сохраняет контроль и прозрачность архитектуры.

В итоге, вдохновение Rust'ом позволило нам построить архитектуру нашего языка так, чтобы сочетать простоту и удобство использования с максимальной безопасностью и производительностью. Мы смогли реализовать язык, в котором можно смело проектировать сложные системы, не жертвуя надежностью и эффективностью, что является одним из ключевых преимуществ перед многими другими языками программирования.

2.3. Общая схема компилятора

Компилятор LANGX представляет собой классический многоступенчатый транслятор:

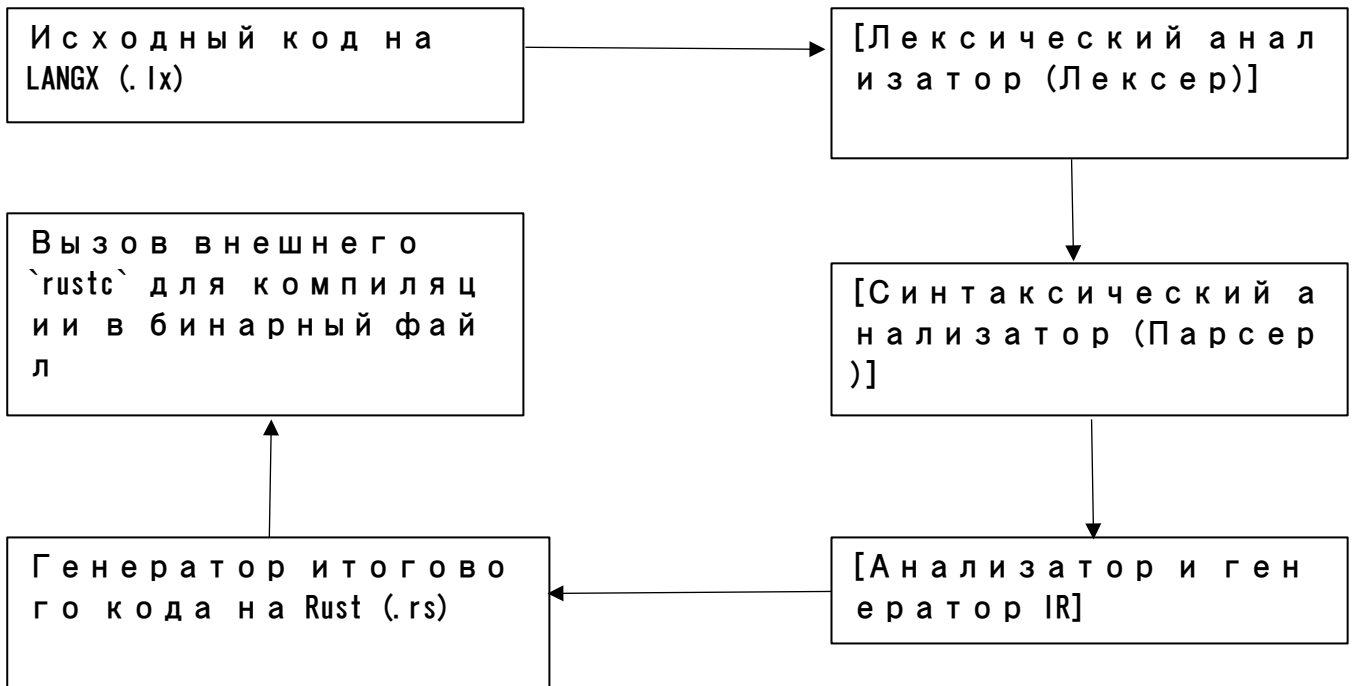


Рис. 7. Схема компилятора

2.4. Этап 1: Лексический анализ

Лексер, реализованный с использованием библиотеки **logos**, выполняет начальный и один из ключевых этапов обработки исходного кода. Его основная задача заключается в последовательном сканировании исходного текста программы и преобразовании потока символов в упорядоченную последовательность токенов. Токены представляют собой элементарные смысловые единицы языка: ключевые слова, идентификаторы, числовые и строковые литералы, операторы, разделители и служебные символы.



Рис. 8. Понятие Лексера

В процессе работы лексер последовательно читает входной текст и сопоставляет фрагменты символов с заранее определёнными правилами и шаблонами токенов. Благодаря использованию **logos**, этот процесс отличается высокой производительностью и строгой детерминированностью, что позволяет эффективно обрабатывать даже большие исходные файлы. Каждому распознанному токеному присваивается конкретный тип, а при необходимости — дополнительная информация, такая как значение литерала или имя идентификатора.

```
src > lexe.rs > Token
1 // Лексер для LangX
2 #[derive(Debug, Clone, PartialEq)]
3 pub enum Token {
4     Fn,
5     Let,
6     Return,
7     If,
8     Else,
9     While,
10    For,
11    Struct,
12    True,
13    False,
14    Print,
15    Ident(String),
16    Int(i64),
17    Float(f64),
18    Void,
19    I32,
20    I64,
21    F32,
22    F64,
23    LParen,
24    RParen,
25    LBrace,
26    RBrace,
27    Comma,
28    Colon,
29    Semicolon,
30    Assign,
31    Plus,
32    Minus,
33    Star,
34    Slash,
35    Eq.
```

Рис. 9. Фрагмент лексера LANGX

2.5. Этап 2: Синтаксический анализ и построение AST

Парсер, реализованный, например, с использованием генератора **lalrpop**, принимает на вход последовательность токенов, сформированную лексером, и выполняет синтаксический анализ программы. Его основная задача заключается в проверке корректности структуры кода в соответствии с формальной грамматикой языка **LANGX** и преобразовании линейного потока токенов в структурированное представление более высокого уровня.



Рис. 10. Парсер

В процессе работы парсер сопоставляет токены с правилами грамматики, определяющими допустимые конструкции языка: выражения, операторы, блоки кода, объявления функций, структур и других сущностей. Если последовательность токенов не соответствует заданным правилам, парсер фиксирует синтаксическую ошибку и формирует диагностическое сообщение, указывая на место и характер нарушения. Это позволяет выявлять ошибки в структуре программы ещё до этапов семантического анализа и выполнения.

Результатом работы парсера является **Абстрактное Синтаксическое Дерево (AST)**. AST представляет программу в виде иерархической структуры узлов, где каждый узел соответствует логической конструкции языка. В отличие от конкретного синтаксиса исходного кода, AST не содержит лишних элементов, таких как скобки, разделители или ключевые слова, если они не несут семантической нагрузки. Вместо этого дерево отражает смысловую структуру программы и взаимосвязи между её компонентами.

```
src > astrs > Type
1  #[allow(dead_code)]
2  #[derive(Debug, Clone)]
3  pub enum Type {
4      I32,
5      I64,
6      F32,
7      F64,
8      Bool,
9      Void,
10     Custom(String),
11 }
12
13 #[derive(Debug, Clone)]
14 pub enum Expr {
15     Int(i64),
16     Float(f64),
17     Bool(bool),
18     Ident(String),
19
20     String(String),
21
22     BinaryOp {
23         left: Box<Expr>,
24         op: String,
25         right: Box<Expr>,
26     },
27
28     Call {
29         func: String,
30         args: Vec<Expr>,
31     },
32
33     Print(Box<Expr>),
34
35     StructInit {
```

Рис. 11. Фрагмент Абстрактного Синтаксического Дерева LANGX

4.5. Этап 3: Генерация промежуточного представления и кодогенерация

Сердцем компилятора является модуль семантического анализа и трансляции, который последовательно обходит Абстрактное Синтаксическое Дерево (AST) и выполняет ключевые этапы проверки и преобразования программы. На этом уровне исходный код на языке LANGX перестаёт быть просто корректным с точки зрения синтаксиса и начинает анализироваться с позиции смысла, логики и корректности используемых конструкций.

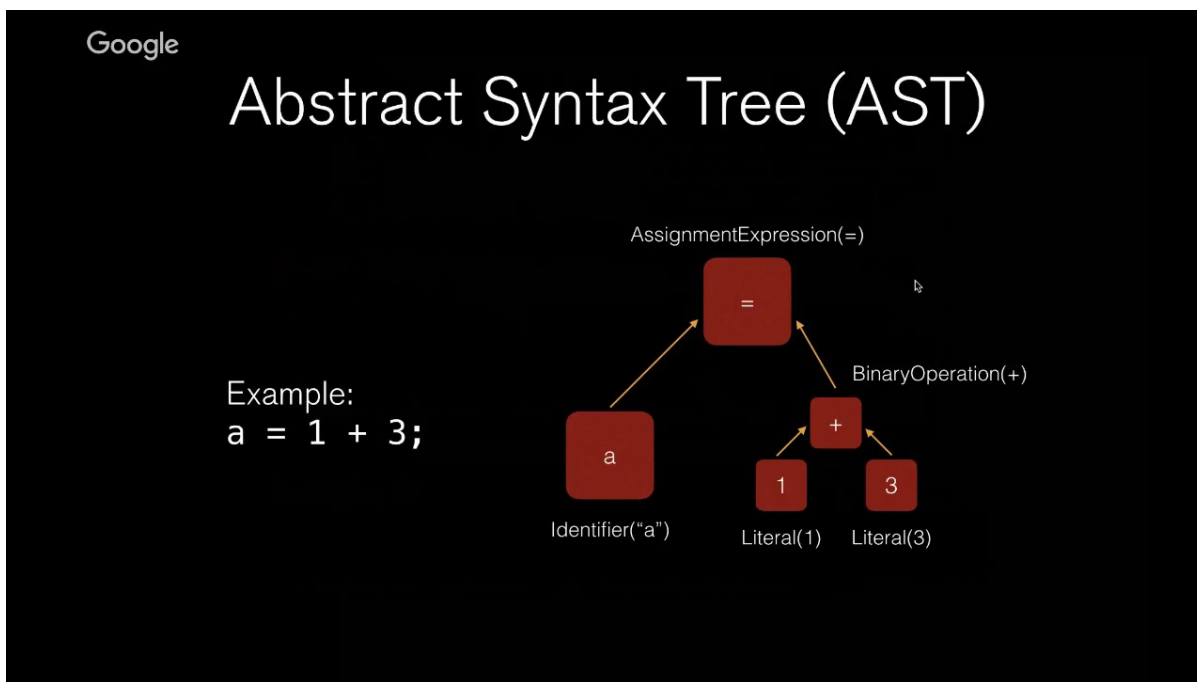


Рис.12. Пример Абстрактного Синтаксического Дерева

В первую очередь данный модуль отвечает за семантическую проверку. Во время обхода AST осуществляется контроль соответствия типов, проверка корректности операций над данными, анализ областей видимости и разрешение имён. Компилятор убеждается, что переменные объявлены до использования, функции вызываются с правильным количеством и типами аргументов, а возвращаемые значения соответствуют заявленным типам. На этом же этапе выявляются логические ошибки, которые невозможно обнаружить на уровне синтаксиса, но которые критичны для корректной работы программы.

Далее выполняется построение и сопровождение таблиц символов, в которых хранится информация о переменных, функциях, типах и других сущностях программы. Эти таблицы позволяют компилятору отслеживать контекст выполнения, корректно обрабатывать вложенные блоки и обеспечивать строгую типовую дисциплину. Такой подход создаёт прочную основу для безопасной и предсказуемой трансляции кода.

```

src > codegen.rs > ...
14 impl CodeGen {
60     for stmt in stmts {
61         match stmt {
62             Stmt::FunctionDef(func) => {
63                 for s in &func.body {
64                     match s {
65                         Stmt::Expr(expr) => match expr {
66                             Expr::Int(n) => {
67                                 let val = builder.ins().iconst(types::I64, *n);
68                                 let puts_ref = module.declare_func_in_func(puts_id, builder.func);
69                                 builder.ins().call(puts_ref, &[val]);
70                             }
71                             Expr::String(s) => {
72                                 let data_id = module
73                                     .declare_data(
74                                         &format!("str_{}", s),
75                                         Linkage::Local,
76                                         false,
77                                         false,
78                                     )
79                                     .map_err(|e| e.to_string())?;
80                                 let mut string_bytes = s.clone().into_bytes();
81                                 string_bytes.push(0);
82                                 let mut data_ctx = CustomDataContext::new();
83                                 data_ctx.define(string_bytes.into_boxed_slice());
84                                 let data_description = cranelift_module::DataDescription::new();
85                                 data_description.define(data_ctx.get_data().to_vec().into_boxed_slice());
86                                 module.define_data(data_id, &data_description).map_err(|e| e.to_string())?;
87                                 let local_id = module.declare_data_in_func(data_id, builder.func);
88                                 let addr = builder.ins().symbol_value(types::I64, local_id);
89                                 let puts_ref = module.declare_func_in_func(puts_id, builder.func);
90                                 builder.ins().call(puts_ref, &[addr]);
91                             }
92                             _ => {}
93                         }
94                     }
95                 }
96             }
97         }
98     }
99 }

```

Рис.13. Код взаимодействия с самим языком и компилятором

- Преобразование объявлений переменных с выводом типов в объявления на Rust (с явными типами или let с выводом).
- Трансляцию управляющих конструкций (if, while, for).
- Преобразование функций LANGX в функции Rust, включая специальную обработку семантики владения для аргументов и возвращаемых значений.
- Генерацию кода для структур данных, определенных пользователем.

Получившийся файл на Rust затем передается компилятору gusted для окончательной сборки.

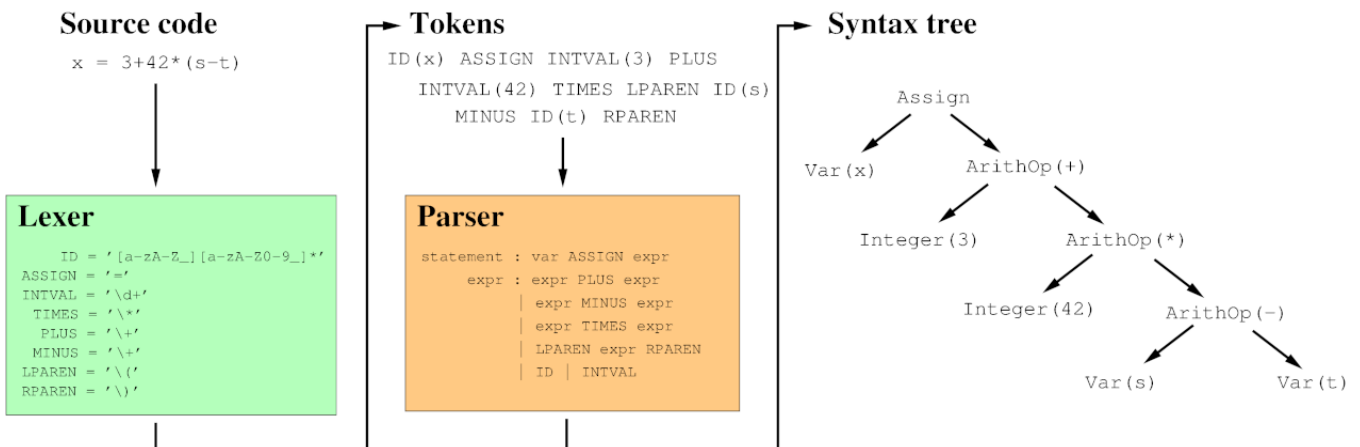


Рис. 14. Основа любого языка программирования

Каждый язык программирования, независимо от области применения и уровня сложности, строится на наборе базовых правил и концепций, которые определяют, как разработчик взаимодействует с компьютером. Эти правила формируют структуру языка, обеспечивают однозначность выполнения программ и позволяют создавать корректные, читаемые и поддерживаемые решения.

Одним из фундаментальных правил любого языка программирования является строгий синтаксис. Синтаксис определяет, каким образом должны быть записаны команды, инструкции и выражения, чтобы компилятор или интерпретатор мог их корректно обработать. Нарушение синтаксических правил приводит к ошибкам, поэтому соблюдение структуры кода является обязательным требованием при разработке программ.

Следующим важным аспектом являются переменные и типы данных. Большинство языков программирования требуют явного или неявного объявления переменных, а также определения типа данных, с которыми они работают. Это правило позволяет управлять памятью, предотвращать логические ошибки и повышать надёжность программ. Типы данных задают, какие операции допустимы над конкретными значениями и как они будут интерпретироваться системой.

Неотъемлемой частью любого языка являются управляющие конструкции: условные операторы, циклы и механизмы ветвления. Они определяют логику выполнения программы, позволяя принимать решения, повторять действия и управлять потоком выполнения кода. Правильное использование этих конструкций — основа алгоритмического мышления и эффективного программирования.

Также важную роль играют функции и модули. В большинстве языков программирования принято разделять код на логические блоки, чтобы повысить его читаемость, повторное использование и удобство сопровождения. Это правило способствует структурированию программ и облегчает совместную разработку, особенно в крупных проектах.

Отдельного внимания заслуживают правила именования и форматирования кода. Хотя они не всегда строго обязательны с точки зрения компиляции, соблюдение единых соглашений повышает понятность программ и облегчает работу с кодом как для самого автора, так и для других разработчиков. Читаемость и ясность — важные качества качественного программного продукта.

Наш язык программирования был разработан с учётом всех этих базовых принципов, но при этом мы стремились упростить и логически выстроить основные правила. В основе нашего языка лежит понятный и минималистичный синтаксис, который снижает порог входа для начинающих и ускоряет процесс разработки. Мы уделили особое внимание тому, чтобы структура кода была интуитивно ясной и не перегруженной избыточными элементами.

Работа с переменными и данными в нашем языке построена таким образом, чтобы сохранить баланс между гибкостью и безопасностью. Управляющие конструкции реализованы в простой и логичной форме, что позволяет сосредоточиться на решении задачи, а не на сложности языка. Функции и логическое разделение кода являются обязательной частью философии языка, так как мы считаем модульность и структурированность ключевыми факторами качественного программирования.

Таким образом, базовые правила языков программирования, включая наш собственный, направлены на достижение одной цели — создание понятного, надёжного и эффективного инструмента для решения задач. Наш язык продолжает общие традиции программирования, при этом предлагая собственный подход, основанный на простоте, практичности и уважении к разработчику.

3. Синтаксис LANGX

3.1. Базовые конструкции и вывод типов

```
rust
//
let x = 42 // Компилятор выводит тип i32
let y: f64 = 3.14 // Явная аннотация типа
let name = "Hello" // Выводится тип &str

// Генерируемый Rust-код (приблизительно)
let x: i32 = 42;
let y: f64 = 3.14;
let name: &str = "Hello";
```

3.2. Функции и модульность

```
rust
//
func factorial(n: i32) -> i32 {
  if n <= 1 {
    return 1
  }
  return n * factorial(n - 1)
}

func main() {
  let result = factorial(5)
  println(result) // Специальная встроенная функция
}

rust
// Соответствующий Rust-код
fn factorial(n: i32) -> i32 {
  if n <= 1 {
    return 1;
  }
  return n * factorial(n - 1);
}

fn main() {
  let result: i32 = factorial(5);
  println!("{}", result);
}
```

3.3. Управление памятью и безопасность

LANGX заимствует у Rust концепцию **владения** для обеспечения безопасности памяти на этапе компиляции без сборщика мусора.

```
rust
//                                     LANGX                                     (концептуально)
struct      Point      {      x:      f64,      y:      f64      }

func use_point(p: Point) { // p переходит во владение функции
//                               ...                               использование p
} // p и его данные удаляются здесь

func main() {
let pt = Point { x: 1.0, y: 2.0 }
use_point(pt) // Владение pt передаётся функции
// println(pt.x) // Ошибка компиляции! pt больше не доступен.
}
```

При разработке нашего языка программирования особое внимание было уделено вопросам управления памятью и безопасности, так как именно эти аспекты во многом определяют надёжность и устойчивость программных систем. В качестве ориентира в данной области для нас послужил язык Rust, который на сегодняшний день считается одним из наиболее показательных примеров безопасного и эффективного подхода к работе с памятью.

Мы позаимствовали у Rust ключевую концепцию строгого контроля владения данными. В основе этого подхода лежит идея того, что каждый участок памяти имеет единственного владельца, отвечающего за его жизненный цикл. Такой механизм позволяет избежать целого класса распространённых ошибок, включая утечки памяти, двойное освобождение и обращение к уже освобождённым данным. В нашем языке эта модель была адаптирована и интегрирована таким образом, чтобы сохранить высокий уровень безопасности без чрезмерного усложнения синтаксиса.

Ещё одним важным элементом, заимствованным из Rust, стала система заимствований. Она регулирует доступ к данным через строгие правила чтения и изменения, не допуская одновременного небезопасного использования памяти. Благодаря этому компилятор способен обнаруживать потенциально опасные ситуации на этапе компиляции, а не во время выполнения программы. Такой подход существенно повышает надёжность кода и снижает вероятность критических ошибок в рабочей среде.

Отдельного внимания заслуживает ориентация на безопасность без использования сборщика мусора. Как и в Rust, в нашем языке управление памятью осуществляется на этапе компиляции, что позволяет достигать высокой производительности и предсказуемого поведения программ. Мы сознательно отказались от автоматического сборщика мусора, сделав ставку на строгие правила владения и времени жизни объектов, поскольку считаем этот подход более подходящим для системного и высоконагруженного программирования.

Кроме того, мы переняли философию предотвращения ошибок по умолчанию. В Rust многие потенциально небезопасные операции требуют явного подтверждения со стороны разработчика. Аналогичный принцип реализован и в нашем языке: разработчик должен осознанно

обозначать места, где допускается снижение уровня безопасности, что формирует ответственное отношение к коду и архитектуре системы.

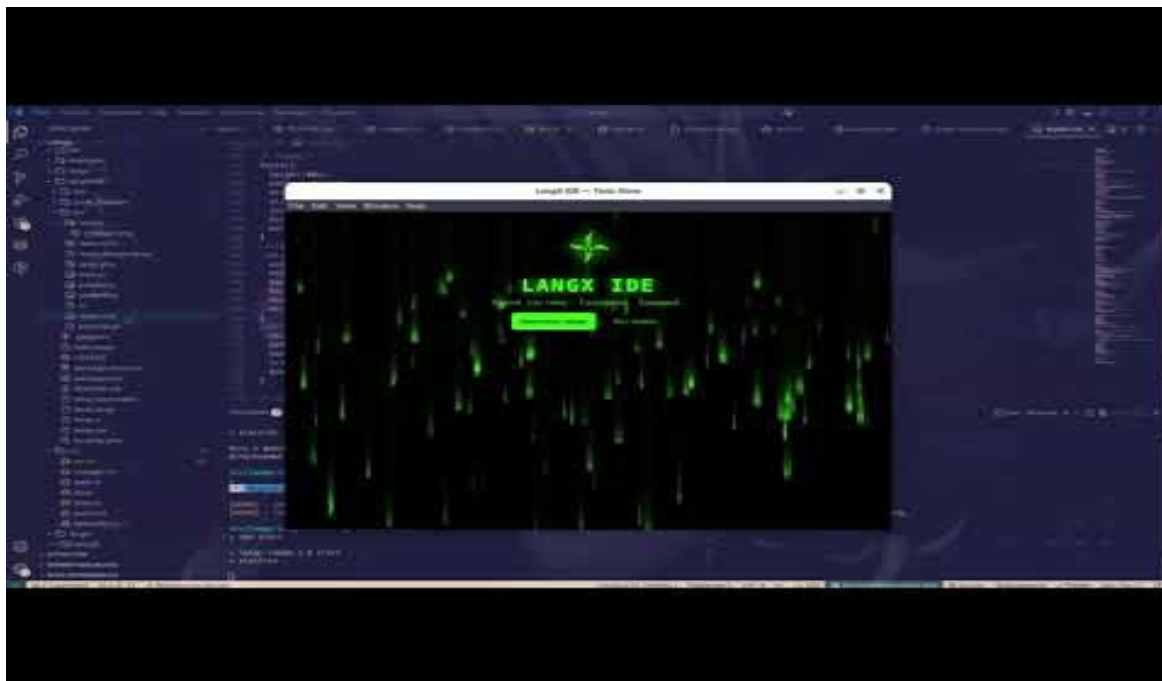
В результате заимствование и адаптация концепций управления памятью и безопасности из Rust позволили нам заложить прочный фундамент для нашего языка программирования. Мы объединили строгую модель безопасности с более доступным и понятным подходом, стремясь создать инструмент, который обеспечивает надёжность на уровне системных языков и при этом остаётся удобным для повседневной разработки.



Рис. 15. Интерфейс LANGX (Начальный экран)

4. План разработки и текущий статус

4.1. Что уже реализовано (Прототип, v0.1.0):



- Лексический анализатор для базового набора токенов.
- Парсер для основных конструкций: переменные, арифметические и логические выражения, управляющие конструкции `if/else`, `while`.
- Генерация простого Rust-кода для вышеуказанных конструкций.
- Интеграция с `cargo` для сборки итогового проекта.

4.2. Ближайшие цели (v0.2.0 - v0.5.0)

- Реализация функций и механизма возврата значений.
- Добавление составных типов данных: кортежи, структуры.
- Введение модульной системы.
- Базовые замыкания (лямбда-функции).
- Разработка минимальной стандартной библиотеки (ввод/вывод, основные операции).

4.3. Долгосрочная перспектива

- Реализация системы трейтов/интерфейсов по образцу Rust.
- Углубленная работа над системой владения и заимствования, адаптированной под упрощённый синтаксис.
- Создание Language Server Protocol (LSP) для поддержки в редакторах кода.
- Разработка REPL (Read-Eval-Print Loop) для интерактивного изучения языка.

Заключение

Немалую роль в эволюции языков сыграет и развитие ИИ. Будущее кодирования увидит появление более специализированных языков машинного обучения, таких как Julia и Swift, которые изменят ландшафт разработки софта. Кроме того, роль машинного обучения и искусственного интеллекта в оптимизации кода, предиктивном обслуживании сократит разрыв между данными и кодом.

Поскольку мир сталкивается с экологическими проблемами, технологическая индустрия продолжит прилагать усилия по обеспечению устойчивости для снижения потребления энергии. Экологичное кодирование или разработка зеленого программного обеспечения подразумевает и написание энергоэффективного кода, который позволит снизить потребление ресурсов. На данный момент наиболее энергоэффективными языками программирования являются C, Rust, C++, Ada, Java, Pascal.

Эволюция языков программирования является свидетельством неустанного стремления к лучшим инструментам для решения сложных задач. С первых дней машинного кода и языка ассемблера до современной эры высокоуровневых мульти-парадигменных языков каждый шаг приближал нас к более продуктивным, выразительным и мощным способам общения с компьютерами.

Проект LANGX представляет собой практическую попытку исследовать дизайн языков программирования и реализацию компиляторов. Он демонстрирует, как современные инструменты (Rust, LLVM) позволяют небольшим командам сосредоточиться на высокоуровневых задачах, делегируя низкоуровневую оптимизацию проверенным технологиям. Этот проект — не только о создании языка, но и об обучении. Мы прошли полный цикл: от формулировки философии и проектирования грамматики до написания лексера, парсера и кодогенератора. LANGX является живым доказательством того, что амбициозные задачи в области системного программирования становятся всё более доступными для тех, кто обладает целеустремленностью и желанием учиться.

Статистика проекта:

- Время активной разработки: 3 месяца.
- Язык реализации: Rust.
- Количество основных крейтов: 5 (logos, lalrpop, clap и др.).
- Объём кода компилятора: ~4000 строк.

Список использованных ресурсов

1. Книги и статьи:

- a. Торбен Эгхольм Могенсен. «Компиляторы: принципы, технологии и инструменты» (Dragon Book).
- b. Стив Клабник, Кэрол Николс. «Язык программирования Rust» (The Rust Programming Language Book).
- c. Статьи о дизайне языков: история C++, Python, Go, Rust.

2. Технологии и библиотеки:

- a. Официальная документация по языку Rust: <https://www.rust-lang.org/>
- b. Документация крейта logos: <https://docs.rs/logos>
- c. Документация крейта lalrpop: <https://docs.rs/lalrpop>
- d. LLVM проект: <https://llvm.org/>

3. Открытые образовательные ресурсы:

- a. Курс «Компиляторы» на Coursera от Стэнфорда.
- b. Репозитории open-source компиляторов (например, rustc, zig).