ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО
ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ «НАЦИОНАЛЬНЫЙ
ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»
**Факультет Санкт-Петербургская школа
физико-математических и компьютерных наук**

Чижова Мария Александровна

Исследование реализации виртуальных машин для встраиваемых языков в языке программирования Go

Выпускная квалификационная работа - БАКАЛАВРСКАЯ РАБОТА
по направлению подготовки
01.03.02 Прикладная математика и информатика
образовательная программа «Прикладная математика и информатика»

Рецензент:
ООО «Техкомпания Хуавей»,
эксперт
М. А. Беляев

Руководитель:
Кандидат физико-математических наук, доцент,
департамент информатики
Д.Н. Москвин

Санкт-Петербург
2023

Содержание

Встраиваемые языки программирования представляют собой языки программирования, написанные внутри другого языка и предназначенные для решения определенных задач. При разработке таких языков важно обеспечить высокую производительность, и создание эффективной виртуальной машины является ключевой задачей.

На данный момент не проводились исследования, сравнивающие различные виртуальные машины для встраиваемых языков программирования в язык Go. Однако есть некоторые бенчмарки, основанные на сравнении скорости работы встраиваемых языков. В таких сравнениях обычно рассматриваются языки, содержащие различные оптимизации, и часто такие сравнения могут быть предвзятыми в пользу автора. Большинство встраиваемых языков используют виртуальную машину, использующую обход абстрактного синтаксического дерева, но есть и некоторые реализации, использующие стековую виртуальную машину.

Это исследование направлено на сравнение реализаций виртуальных машин для встраиваемых языков в Go. В данной работе было реализовано пять различных виртуальных машин: виртуальная машина, использующая обход АСД, три стековые (с одним и с несколькими стеками, и с использованием рефлексии) и регистровая виртуальная машина, и проведен сравнительный анализ их скорости работы.

Ключевые слова: встраиваемые языки, язык программирования Go, виртуальные машины, компиляторы, анализ производительности

Embedded programming languages are specialized languages that are written within another language and optimized to solve specific problems. Ensuring high performance is crucial during the development of such languages, and creating an efficient virtual machine is a key objective.

There have been no studies comparing different virtual machines for embedded programming languages in Go. However, there are some benchmarks available that compare the performance of embedded languages. These comparisons usually involve languages with various optimizations and often can be biased in favor of the author. Most embedded languages use tree traversal, but there are some implementations that use a stack-based virtual machine.

This study aims to compare different implementations of virtual machines for embedded programming languages in Go. In this work, five different virtual machines were implemented: tree traversal, three stack-based machines (with single and multiple stacks, as well as reflect-based), and a register-based virtual machine. The study includes a comparative analysis of their performance.

Key words: embedded languages, Go programming language, virtual machines, compilers, performance analysis

Введение в область

В последние годы язык программирования Go[31] (также известный как Golang) приобрел большую популярность благодаря своей простоте, эффективности и удобству использования. Разработанный в компании Google в 2007 году, он был выпущен в открытый доступ в ноябре 2009 года и с тех пор привлек множество разработчиков, которые вносят свой вклад в его развитие, создавая новые инструменты и библиотеки, расширяющие его функциональность.

Одна из областей, в которой язык Go является популярным, является реализация встраиваемых языков программирования. Встраиваемые языки программирования – это языки программирования, написанные внутри другого языка и предназначенные для решения определенных задач, таких как фильтрация, конфигурация, бизнес-логика и другие. Они позволяют решать эти задачи эффективным и простым способом.

В отличие от языков общего назначения, таких как C++ или Java, встраиваемые языки программирования обычно не являются Тьюринг-полными, то есть они не позволяют зацикливать программы, поскольку могут выполнить только ограниченные выражения.

Встраиваемые языки программирования широко применяются в различных проектах и компаниях. Например, они могут быть использованы, когда заранее неизвестно, как будет выглядеть выражение, или когда мы хотим, чтобы выражения были конфигурируемыми. К примеру, у нас могут быть данные, проходящие через наше приложение, и мы хотим, чтобы пользователи могли указать определенные проверки, которые нужно

выполнить перед сохранением в базе данных. Также, в случае, если мы хотим написать инфраструктуру мониторинга, которая собирает множество метрик и оценивает несколько выражений, чтобы увидеть, следует ли выдавать предупреждения о каких-либо метриках. При этом условия для выдачи предупреждений могут быть различны для каждого монитора.

Встраиваемые языки программирования позволяют быстро вносить изменения в систему, тем самым избегать долгих ожиданий при развертывании всей системы. Кроме того, встраиваемые языки понятны и просты в использовании, что делает их идеальным выбором для экспертов в области, не являющихся программистами.

Постановка задачи

Одной из главных задач при разработке встраиваемых языков программирования является создание виртуальной машины, которая будет отвечать за выполнение программ на данном языке.

Большинство существующих решений [11, 12, 14, 17, 18, 23] используют виртуальную машину, использующую обход абстрактного синтаксического дерева (АСД), из-за простоты ее реализации и особенностей языка Go[33]. Также существуют некоторые решения, использующие стековую виртуальную машину [16, 19, 24].

На данный момент не проводились исследования, сравнивающие различные виртуальные машины для встраиваемых языков программирования в Go. Однако есть некоторые бенчмарки [20], основанные на сравнении скорости работы встраива-

емых языков, которые содержат разное количество оптимизаций. Некоторые языки могут иметь больше оптимизаций, в то время как другие могут их вообще не иметь. Кроме того, существующие сравнения часто являются предвзятыми в пользу автора, поскольку для сравнения можно выбрать те выражения, на которых язык проявляет себя наилучшим образом, что искажает результаты сравнения. Это может затруднить выбор оптимальной архитектуры для разработки виртуальной машины.

В данной работе мы рассмотрим различные виды виртуальных машин во встраиваемых языках в Go и сравним их производительность. Это позволит выявить наиболее эффективные способы реализации виртуальных машин для встраиваемых языков программирования и, в последствии, улучшить скорости работы приложений, использующих эти языки.

Цель и основные задачи

Целью данной работы является сравнение производительности реализаций виртуальных машин для встраиваемого языка программирования в язык программирования Go.

Для достижения этой цели были поставлены следующие задачи:

1. Определение синтаксиса встраиваемого языка программирования, для которого будет проведено сравнение виртуальных машин.

2. Реализация лексера и парсера для этого языка.

3. Реализация различных виртуальных машин:

- Виртуальная машина, использующая обход АСД.
- Стековая виртуальная машина с различными модификациями: с одним/несколькими стеками, использующая рефлексию.
- Регистровая виртуальная машина.

4. Написание бенчмарков и проведение сравнения производительности реализаций виртуальных машин.

Структура работы

В первой главе представлен обзор существующих встраиваемых языков программирования и подходов к реализации виртуальных машин.

Во второй главе описан предложенный подход реализации и сравнения виртуальных машин.

В третьей главе описаны технические подробности реализации нашего подхода.

В четвертой главе проведено сравнение производительности реализованных виртуальных машин и их оценка.

В заключении приведен анализ проделанной работы и представлены перспективы для дальнейшей деятельности.

1 Обзор литературы

1.1 Виртуальные машины

Термин ”виртуальная машина” имеет множество значений. В данной дипломной работе термин ”виртуальная машина” будет

использоваться в качестве абстрактной вычислительной машины, определяющей исполняемое представление программы и архитектуру команд.

1.1.1 Виртуальная машина, использующая обход АСД

Виртуальная машина, использующая обход АСД[3, 22] – это программа, которая рекурсивно обходит все узлы АСД и выполняет инструкции, находящиеся в каждом узле. Абстрактное синтаксическое дерево (АСД) [26] – древовидная структура данных, представляющая структуру программного кода, где каждый узел дерева соответствует определенной конструкции языка программирования, а листья дерева содержат конкретные значения и идентификаторы переменных.

Одним из недостатков этого подхода является то, что каждый узел абстрактного синтаксического дерева хранит каждое из своих поддеревьев в виде отдельных указателей, что приводит к необходимости выполнять множество разрозненных доступов к памяти, которые вызывают промахи кэша. Поэтому данные виртуальные машины работают медленнее, чем другие виртуальные машины, но при этом они отличаются относительной простотой в реализации, что является их преимуществом.

1.1.2 Стековая виртуальная машина

Стековая виртуальная машина[2, 22, 26] использует байт-код для выполнения программы. Байт-код представляет собой сжатое представление абстрактного синтаксического дерева с определенным форматом и набором инструкций (опкодов), которые

отличаются для различных языков программирования. Хотя опкоды часто напоминают те, что можно найти в языках ассемблера, байт-код не является нативным машинным кодом и не может быть выполнен напрямую процессором или операционной системой.

Стековая виртуальная машина использует стек для хранения и манипуляции данными. Для выполнения операции необходимо получить данные из стека, выполнить операцию и положить результат обратно в стек.

По сравнению с виртуальной машиной, использующей обход АСД, стековая виртуальная машина имеет более предсказуемый паттерн доступа к памяти. Кроме того, наличие байт-кода позволяет его оптимизировать.

Другим преимуществом стековой архитектуры является отсутствие необходимости указывать адрес операнда, что упрощает формат байт-кода и облегчает процесс генерации кода в целом. Обычно к каждой инструкции предоставляется не более одного аргумента. Кроме того, общий размер кода реализации стековой архитектуры обычно гораздо меньше по сравнению с регистровой.

Архитектура на основе стека широко применяется в виртуальной машине Java (JVM) для выполнения байткодов Java, а также в CPython для интерпретации исходных кодов Python.

1.1.3 Регистровая виртуальная машина

Регистровая машина[26] использует альтернативный подход к хранению данных, основанную на использовании выделенного набора ячеек памяти с фиксированными именами-номерами,

называемыми регистрами. Регистры используются для эмуляции архитектуры физического компьютера.

Регистровая виртуальная машина, как и стековая, использует байт-код для исполнения программы, но байт-код отличается от байт-кода стековой виртуальной машины тем, что требует явного указания адреса операндов. Это позволяет сократить размер байткода, что обычно ведет к увеличению скорости исполнения программы. Например, в коде, который складывает две переменные (см. Рис. 1), байт-код стековой виртуальной машины содержит 4 инструкции, в то время как байт-код стековой содержит всего одну.

```
# Код              Стековая          Регистровая
  a = 1;           load <a>          add <a> <b> <c>
  b = 2;           load <b>
  c = a + b;       add
                   store <c>
```

Рис. 1: Пример кода, который складывает две переменные. Рядом с ним показан соответствующий пример байт-кода для стековой и регистровой виртуальных машин.

К тому же байт-код регистровой машины позволяет провести оптимизации, которые не могут быть выполнены при стековом подходе. Например, несколько раз встречающееся выражение при регистровом подходе может быть вычислено лишь однажды и сохранено в регистре для последующего использования, что экономит время необходимое для пересчета выражения.

Однако реализация компилятора для регистровой виртуальной машины может потребовать гораздо больше усилий и вре-

мени, чем для стековой виртуальной машины.

Например, в новых версиях языка программирования Lua [15] удалось улучшить скорость работы за счет перехода со стековой на регистровую архитектуру.

1.2 Способы диспетчеризации инструкций

Помимо выбора архитектуры в создании виртуальной машины важен выбор способа диспетчеризации инструкций. Диспетчеризация заключается в извлечении опкода инструкции и поиске соответствующего сегмента, который реализует инструкцию перед ее выполнением.

1.2.1 Оператор switch

Один из самых простых и распространенных способов диспетчеризации - использование оператора switch с множеством case в цикле выполнения виртуальной машины[8]. Однако, хотя оператор switch может казаться наиболее очевидным выбором, на самом деле существуют и другие, более эффективные способы диспетчеризации, особенно при большом количестве вариантов case.

1.2.2 Таблицы переходов

Таблицы переходов, также известные как jump tables[28] - это массив указателей на функции, которые обрабатывают операции. Индекс массива соответствует номеру операции, поэтому поиск функции для обработки операции осуществляется простым обращением к ячейке массива. Это более эффективный

способ, чем использование оператора switch, поскольку позволяет избежать необходимости перебирать значения в таблице. Однако, для хранения таблицы требуется больше памяти.

1.2.3 Шитый код

1. **Прямой шитый код**
   Для улучшения скорости работы используется прямой шитый код[4]: опкоды инструкций виртуальной машины заменяются на адреса их реализации в виртуальной машине. Такой подход позволяет вызывать функции непосредственно из кода операции, избегая дополнительных обращений к памяти для поиска указателя на функцию, что ускоряет работу программы. Однако, этот подход требует дополнительного этапа перевода и увеличивает размер кода для представления инструкций виртуальной машины.

2. **Косвенный шитый код**
   Косвенный шитый код[7] использует дополнительный уровень косвенной адресации, чтобы сэкономить место для кода виртуальной машины. В отличие от прямого шитого кода, опкод инструкции виртуальной машины указывает на структуру. Структура имеет указатель адреса на реальную реализацию инструкции виртуальной машины и непосредственные значения и/или операнды для выполнения инструкции. Фактически, код виртуальной машины будет преобразован в список указателей на структуры. Повторяющиеся операнды и/или непосредственные значения будут способствовать экономии места. Однако, из-за уровня косвенной адресации, этот подход не так эффективен, как предыдущий.

В языке Go в настоящий момент нет возможности использования шитого кода.

1.3 Встраиваемые языки программирования

Среди широко используемых встраиваемых языков программирования в Go можно выделить следующие: expr[19], CEL-go[11], govaluate[18] и starlark[12].

1.3.1 Expr

Expr[19] — это язык выражений для Go, который представляет простой, быстрый и расширяемый способ использовать выражения внутри конфигураций для более сложной логики (см. Рис. 2). Expr широко используется в различных компаниях. В Aviasales Expr используется в качестве движка бизнес-правил, в TikTok - в системе поддержки клиентов. Компания Philips Labs использует Expr в Tabia, инструменте для сбора информации о характеристиках кодовых баз, а компания Argo использует в Argo Rollouts и Argo Workflows для Kubernetes.

В начале своего существования Expr использовал виртуальную машину, использующую обход абстрактного синтаксического дерева, но в скором времени перешел на стековую виртуальную машину, что позволило значительно улучшить производительность.

```
# Получить специальную цену, если:
user.Group in ["good_customers", "collaborator"]

# Продвинуть статью на главную страницу, когда:
len(article.Comments) > 100 and article.Category not in ["misc"]

# Отправить оповещение, когда:
product.Stock < 15
```

Рис. 2: Пример использования языка expr.

1.3.2 CEL-go

CEL-go (Common Expression Language)[11] - это простой и безопасный язык, который реализует общую семантику для оценки выражений, упрощая взаимодействие различных приложений. Кроме того, CEL-go является protobuf-ориентированным языком [32], что может стать как достоинством, так и недостатком.

CEL-go широко используется в различных инструментах и платформах, таких как Google IAM и Kubernetes и другие. В Google IAM, CEL-go используется для определения политик безопасности, а в Kubernetes - для определения правил валидации, политик и других ограничений или условий.

Реализация CEL-go основана на виртуальной машине с использованием обхода абстрактного синтаксического дерева.

1.3.3 govaluate

govaluate[18] - встраиваемый язык, который обеспечивает поддержку для вычисления произвольных C-подобных арифметических и строковых выражений. Он широко используется в различных компаниях, включая компанию LogicMonitor, где он используется в системе мониторинга инфраструктуры.

Реализация govaluate также основана на виртуальной машине с использованием обхода абстрактного синтаксического дерева.

1.3.4 starlark

Starlark (ранее известный как Skylark)[12] - это диалект Python, предназначенный для использования в качестве языка конфигураций. Изначально он разрабатывался для системы сборки Bazel[10] в качестве нотации для своих BUILD файлов, а также для своего макроязыка, который расширяет Bazel с помощью пользовательской логики для поддержки новых языков и компиляторов. Starlark в настоящее время используется в различных инструментах и системах, включая Buck - систему сборки, разработанную Facebook, Copybara - инструмент для трансформации и перемещения кода между репозиториями, а также lucicfg из Chromium CI, который генерирует файлы конфигурации на основе Starlark.

В его реализации используется виртуальная машина, использующая обход АСД. Создатели языка проводили эксперименты[33] с использованием виртуальной машины, исполняющей байт-код, но на момент создания языка Go имел следующие ограничения, которые препятствовали улучшению производи-

тельности:

1. Go не генерирует "$computed\ goto$" для оператора $switch$[29]. Основной цикл виртуальной машины, исполняющей байт-код, представляет собой цикл for вокруг оператора $switch$ с десятками или сотнями кейсов, и скорость обработки каждого кейса сильно влияет на общую производительность. На момент создания starlark оператор $switch$ генерировал бинарное дерево упорядоченных сравнений, требующее нескольких ветвлений вместо одного.

2. Анализ побега (escape analysis) компилятора Go предполагает, что базовый массив от выделения $make([]Value, n)$ всегда "убегает"[30]. Поскольку стек операндов виртуальной машины имеет переменную длину, его необходимо выделять с помощью $make$, что приводит к дополнительным затратам при каждом вызове функции. Однако это может быть оправдано за счет амортизации одного очень большого выделения стека на множество вызовов. Более проблемными являются затраты на дополнительные барьеры записи сборщика мусора, возникающие при каждой операции виртуальной машины: каждый промежуточный результат сохраняется в стеке операндов виртуальной машины, который находится в куче. В отличие от этого, промежуточные результаты при использовании подхода с виртуальной машиной, использующую обход АСД, никогда не сохраняются в куче.

1.4 Существующие сравнения

Исследования виртуальных машин для встраиваемого языка в языке программирования Go ранее не проводились. Однако были обнаружены некоторые бенчмарки, такие как [20], которые проводили сравнение различных встраиваемых языков программирования в Go. Важно отметить, что эти сравнения были основаны на сравнении языков, которые содержат разное количество оптимизаций. Некоторые языки могут обладать бóльшим количеством оптимизаций, в то время как другие могут вообще их не иметь.

Кроме того, существующие сравнения часто являются предвзятыми в пользу автора, поскольку для сравнения можно выбрать те выражения, на которых язык проявляет себя наилучшим образом, что искажает результаты сравнения. Также обычно рассматривается очень ограниченный набор выражений, что не позволяет получить полную оценку для выбора оптимальной архитектуры для разработки виртуальной машины.

Кроме того, существует достаточно много исследований[9, 35] виртуальных машин в других языках программирования, которые могут предоставить полезную информацию при разработке виртуальных машин в Go. Однако, важно отметить, что наше исследование сосредотачивается на сравнении виртуальных машин для встраиваемых языков в рамках языка программирования Go. Это означает, что результаты исследований, проведенных для других языков программирования, могут не полностью применимы к вашей задаче, так как язык Go имеет свои уникальные особенности.

1.5 Выводы

Проведенный анализ в областях встраиваемых языков программирования и виртуальных машин позволил сделать следующие выводы:

- Существует несколько подходов к реализации виртуальных машин, включая виртуальную машину, использующую обход абстрактного синтаксического дерева, стековую виртуальную машину и регистровую виртуальную машину.
- Виртуальная машина, использующая обход АСД, является наиболее распространенной при реализации встраиваемых языков программирования. Некоторые реализации используют стековую виртуальную машину. Стоит отметить, что создателям языка Expr удалось повысить производительность при переходе на стековую виртуальную машину. Однако, создатели языка Starlark решили не переходить на виртуальную машину, работающую с байт-кодом, из-за некоторых особенностей языка Go.
- Исследований, сравнивающих виртуальные машины для встраиваемых языков в Go, ранее не проводились. Однако найдены некоторые бенчмарки, сравнивающие скорость работы встраиваемых языков. Стоит отметить, что данные бенчмарки сравнивают языки с различными оптимизациями и могут быть предвзятыми в пользу автора.
- Помимо выбора основной архитектуры при создании виртуальной машины также не менее важны другие особенности, которые следует учитывать. Среди них - способ диспетчери-

зации и особенности языка программирования, на котором виртуальная машина реализуется.

2 Предлагаемый подход

Для сравнения различных реализаций виртуальных машин можно использовать существующие встраиваемые языки программирования, которые имеют соответствующие виртуальные машины. Однако такие языки уже содержат большое количество оптимизаций и особенностей, что делает их сравнение затруднительным. В этом случае более реалистичным подходом будет написание виртуальных машин для более простого встраиваемого языка программирования.

Первым шагом при реализации встраиваемого языка программирования в Go является определение его синтаксиса и возможностей.

Далее поскольку виртуальная машина работает не напрямую с исходным кодом программы, а с абстрактным синтаксическим деревом или байт-кодом, то для удобства генерации АСД необходимо реализовать лексер и парсер. Лексер выполняет лексический анализ программы, преобразуя её в токены, которые затем передаются парсеру. Парсер, в свою очередь, выполняет синтаксический анализ, превращая токены в АСД.

Кроме того, для стековых виртуальных машин мы реализуем компилятор. Компилятор – программа, преобразующая АСД в байткод. При создании компилятора для регистровой машины возникает необходимость применения алгоритмов распределения регистров. Для нетривиальных программ всегда возника-

ет проблема нехватки регистров, так как их количество ограничено, что требует отслеживания содержимого каждого регистра в каждый момент времени. Эти сложности делают реализацию компилятора достаточно объемной задачей, которая выходит за рамки данного исследования. В данном исследовании мы ограничимся написанием функций, которые будут генерировать байт-код для нужных выражений, вместо полной реализации компилятора для регистровой виртуальной машины.

Следует отметить, что существует подход, известный как JIT (just-in-time) компиляция[1], при котором операции, заданные байт-кодом не выполняются непосредственно в виртуальной машине, а вместо этого виртуальная машина компилирует байт-код в родной машинный код прямо перед его выполнением. JIT-компиляция является наиболее эффективным способом выполнения программы, так как компиляция в реальном времени позволяет оптимизировать код под конкретную архитектуру процессора. Однако в данной работе мы не рассматриваем этот подход, поскольку JIT-компиляция требует значительных усилий и зависит от архитектуры процессора.

Для реализации и сравнения производительности нами было выбрано 5 виртуальных машин:

1. Виртуальная машина, использующая обход АСД.
2. Одностековая виртуальная машина.
3. Виртуальная машина с несколькими стеками.
4. Рефлексивная виртуальная машина.
5. Регистровая виртуальная машина.

При реализации виртуальных машин мы будем использовать switch диспетчеризацию, описанную в предыдущей главе, поскольку это хотя не самый быстрый подход, но до сих пор широко используемый.

Далее будут реализованы бенчмарки для различных выражений, похожих на используемые в подобных языках, и проведен сравнительный анализ скорости работы виртуальных машин. Кроме того, для проверки корректности всех компонент будут реализованы юнит-тесты.

Выводы по главе

В данной работе предполагается определить синтаксис для встраиваемого языка программирования, для которого будет проведено сравнение виртуальных машин. Затем будут реализованы лексер и парсер для создания АСД. Также для стековых виртуальных машин будет разработан компилятор и для регистровых виртуальных машин функции для генерации соответствующего байт-кода для необходимых выражений для сравнения виртуальных машин. Для оценки производительности виртуальных машин будут реализованы бенчмарки и проведен сравнительный анализ. Корректность работы будет проверена с помощью реализации юнит-тестов.

3 Реализация предлагаемого подхода

3.1 Спецификация языка

В нашем случае встраиваемый язык будет являться языком выражений, то есть языком, который компилирует и выполняет различные выражения. Выражение – это одна строка кода, которая возвращает значение, обычно булево, но не ограничивается этим. Язык будет иметь следующее:

- **Типы данных:**
  - Целые числа: 1, 01
  - Числа с плавающей точкой : 1.2, 1e2, 1.2e-3, 0., .1
  - Булевые значения: true, false
  - Строки: “abc”, ‘abc’
  - Ноль: nil
  - Массивы: [”a”, ”b”, ”c”]
- **Операции:**
  - Арифметические: +, -, *, %, /, ^
  - Сравнения: >, <, ==, >=, <=, !=
  - Логические: and, or, not
- **Переменные:**
  передаются извне:

  ```
  map[string]interface{}{”a”: 1.2, ”b”: 2.3}
  ```
- **Вызовы внешних функций**
  передаются извне:

```
map[string]interface{}{"add": func(a, b int) int { return a + b }}
```

3.2 Лексический анализ и парсинг

Существует различные алгоритмы и стратегии парсинга [6, 13], а также специализированные инструменты для создания парсеров [25, 34]. Мы решили реализовать свой парсер по нескольким причинам. Во-первых, мы хотели иметь полный контроль над парсером. Во-вторых, генераторы парсеров обычно требуют определенных структур грамматики, и понимание их метода работы может быть достаточно сложным, что может привести к потенциальным проблемам. Наконец, наша основная цель заключается в реализации виртуальных машин, а не самого парсера, поэтому мы предпочли упростить этот аспект данной работы. К счастью, наш язык несложный, поэтому создание собственного парсера не является трудной задачей.

Для реализации лексера был использован алгоритм, описанный Робом Пайком[27], в то время как для реализации парсера был выбран алгоритм рекурсивного спуска с учетом приоритета операторов [6, 21].

3.3 Виртуальная машина, использующая обход АСД

Как уже было описано в обзоре литературы, виртуальная машина, использующая обход АСД, рекурсивно обходит все узлы АСД и выполняет соответствующие инструкции. Для обхода дерева используется алгоритм обратного типа обхода (postorder

traversal). На рисунке 3 представлен пример реализации данной виртуальной машины.

```
func Eval(node ast.Node, env interface{}) (interface{}, error) {
    switch node.Type() {
    case ast.NodeBinary:
        return EvalBinary(node, env)
    ...
    }
    return nil, nil
}
```

Рис. 3: Пример реализации виртуальной машины, использующей обход АСД.

Для более ясного понимания процесса обхода дерева можно рассмотреть пример обработки узла $NodeBinary$. В этом случае будет вызван метод $EvalBinary$, который обработает сначала левое поддерево, затем правое поддерево, а после этого применит к ним соответствующий оператор (см. Рис. 4).

```
left, err := Eval(node.(*ast.BinaryNode).Left, env)
right, err := Eval(node.(*ast.BinaryNode).Right, env)
switch node.(*ast.BinaryNode).Operator {
    case "+":
            ...
    case "-":
            ...
}
```

Рис. 4: Пример реализации функции EvalBinary.

3.4 Одностековая виртуальная машина

Перед выполнением программы виртуальной машиной АСД преобразуется в байт-код с помощью компилятора, реализация которого была вдохновлена книгами Торстена Болла[2] и Роберта Нистрома[22]. Данный компилятор одинаковый для всех стековых виртуальных машин, реализованных в данной работе. На рисунке 5 представлен пример кода и соответствующего ему байт-кода. Для хранения констант компилятор использует отдельный массив, а в байт-коде используются соответствующие им индексы.

```
Выражение: foo("arg1", 2, true, [1, 2, 3], a)

Байткод:
0000 OpConstant 0
0003 OpConstant 1
0006 OpTrue
0007 OpConstant 2
0010 OpConstant 3
0013 OpConstant 4
0016 OpArray 3
0019 OpConstant 5
0022 OpConstant 6
0025 OpCall 5

Константы: ["arg1", 2, 1, 2, 3, a, "foo"]
```

Рис. 5: Пример кода и соответствующего ему байт-кода.

В данной виртуальной машине данные обрабатываются при

помощи одного стека, который хранит значения типа $interface\{\}$. В языке программирования Go $interface\{\}$ кодирует три вещи: значение, набор методов и тип сохраненного значения (см. Рис. 6). Когда функция принимает $interface\{\}$ в качестве параметра, передача значения в эту функцию упаковывает значение, набор методов и тип в $interface\{\}$. Соответственно, $interface\{\}$ позволяет нам хранить любой тип данных внутри него. Однако для выполнения операций над извлеченными значениями необходимо проверять их тип и выполнять приведение к нужному типу.

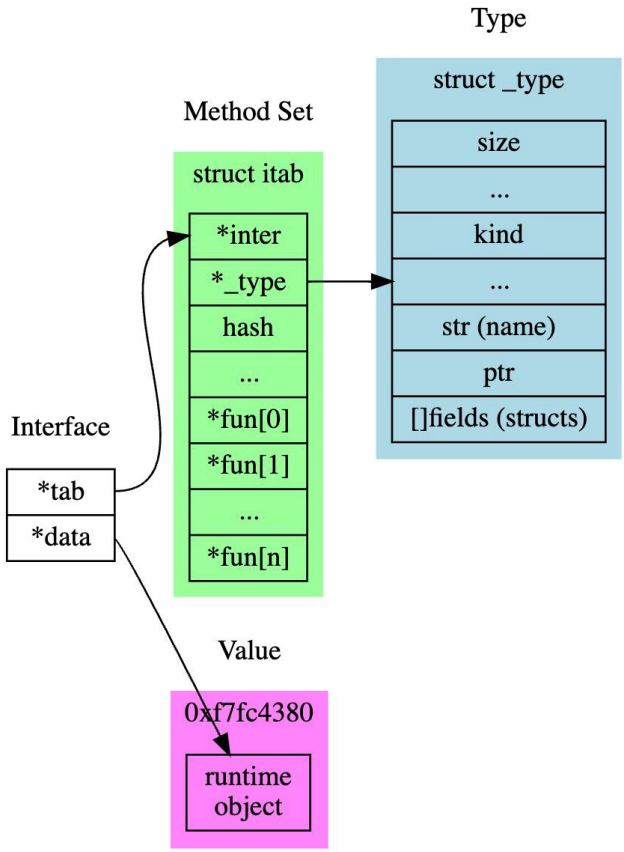

Рис. 6: Структура интерфейса в Go.

На текущий момент виртуальная машина поддерживает 25 инструкций. Большинство из них были выбраны таким образом, чтобы напрямую соответствовать языковым конструкциям. Логические операторы or и and реализованы при помощи инструкций перехода (jump instructions). Кроме того, виртуальная машина принимает окружение, в котором задаются внешние функции и переменные.

3.5 Виртуальная машина с несколькими стеками

Для уменьшения нагрузки на рантайм при приведении типов была реализована виртуальная машина с несколькими стеками. Концепция заключается в использовании дополнительного стека для каждого типа данных. В рамках проведенных экспериментов были добавлены отдельные стеки для строк и целых чисел.

3.6 Рефлексивная виртуальная машина

Рефлексия (reflection) – это механизм, который позволяет программе проверять свое состояние, исследовать типы данных и изменять свою структуру и поведение во время выполнения. В языке Go рефлексия реализуется с помощью пакета $reflect$.

В Go для вызова функции нужно получить соответствующий reflect объект, так как мы заранее не знаем ее тип, чтобы использовать через $interface\{\}$. Для этого мы получаем тип и параметры функции в виде рефлекта. Однако эта дополнительная обработка требует дополнительного времени.

Использование типа $interface\{\}$ на стеке при вызове функций может приводить к дополнительным преобразованиям между $reflect.Value$ и $interface\{\}$. Чтобы избежать этих преобразований, можно использовать напрямую тип $reflect.Value$ на стеке вместо $interface\{\}$. Это может позволить улучшить производительность при вызове функций.

3.7 Регистровая виртуальная машина

Как уже упоминалось в предыдущей главе, для регистровой виртуальной машины компилятор не был реализован, но для сравнения производительности были написаны различные функции, которые генерируют соответствующий байт-код.

Регистровая виртуальная машина имеет 16 регистров, каждый из которых хранит значения типа $interface\{\}$. Аналогично стековой виртуальной машине, константы хранятся в отдельном массиве, а в байт-коде вместо констант хранится их индекс, что также способствует ускорению работы виртуальной машины.

Исходный код доступен на Github[5].

3.8 Выводы и результаты по главе

В данной работе было разработано 5 различных виртуальных машин для встраиваемого языка программирования в Go: виртуальная машина с использованием обхода АСД, три модификации стековых виртуальных машин (с одним, несколькими стеками и использующая рефлексию) и регистровая виртуальная машина. Для упрощения генерации АСД были реализованы лек-

сер и парсер, а для стековых виртуальных машин был разработан компилятор, который преобразует АСД в байт-код.

4 Оценка результатов

4.1 Бенчмарки для сравнения

Бенчмарк - это метод оценки производительности кода, который заключается в многократном выполнении определенного сегмента кода и сравнении каждого результата с определенным стандартом. Для сравнения производительности виртуальных машин были выбраны следующие бенчмарки:

- Сумма от 1 до 200: $1 + 2 + 3 + ... + 200$.
- Чередующаяся сумма: $1 - 2 + 3 - ... - 200$.
- Конкатенация 200 строк: $"a" + "b" + ... + "aa" + "ab" + ...$
- Комбинация сравнения строк и логических операций с булевыми значениями: $("a" \neq "b")\ and/or\ ("a" \geq "bc")$ с 200 сравнениями.
- Вызовы внешних функций с разным количеством аргументов: $foo2(a1, a2) + \cdot + foo10(a1, \ldots, a10)$.
- Вызовы внешних функций с одинаковым количеством аргументов: $foo1(a, b) + \ldots + foo10(y, z)$.
- Сумма внешних переменных: $a1 + \ldots + a100$.
- Выражение с использованием большинства возможностей языка приближенное к выражениям, используемым во встраиваемых языках на практике:

```
foo(parameter1, 3.1) > 5 and 2 > 1.5 and 'something' != 'nothing'
or var + 1000 / 2 >= (80 * 100 % 2)

Окружение: var env = map[string]interface{}{
    "var":  5.0,
    "parameter1": 10.1,
    "foo":  func(a, b float64) float64 { return a * b },
}
```

Все бенчмарки были выполнены на процессоре Apple M1 Pro с 16 GB оперативной памяти. Каждая виртуальная машина не использовала никаких оптимизаций на этапе компиляции. Go включает в себя встроенные инструменты для написания бенчмарков в пакете *testing*. Все результаты измеряются в наносекундах. Поскольку бенчмарки были запущены большое количество раз, то конечный результат представлен в виде среднего значения. Результаты представлены ниже:

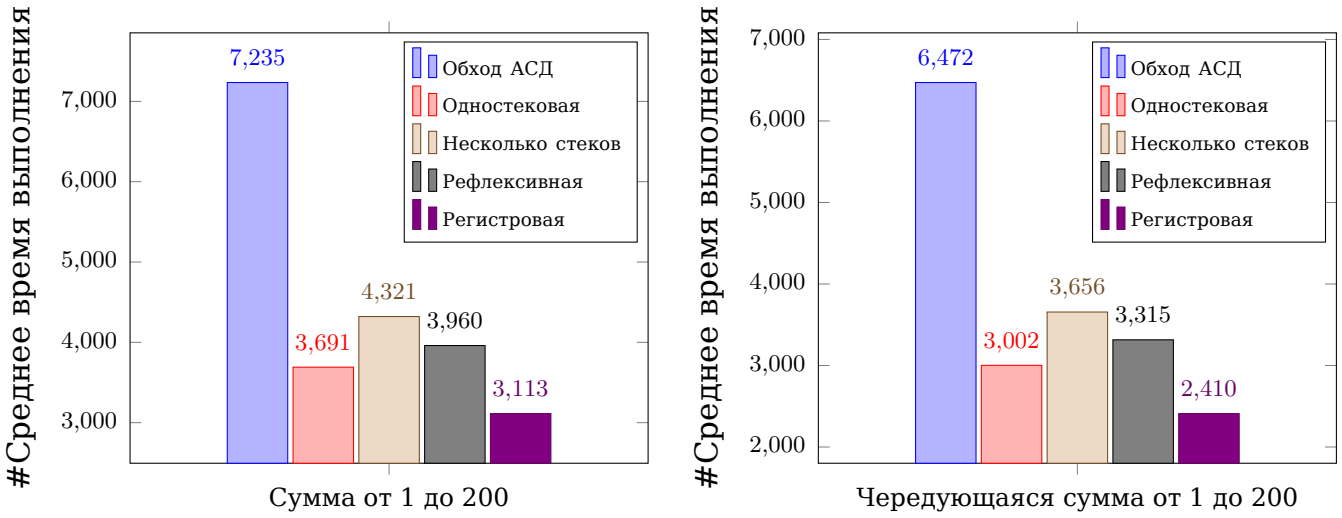

Рис. 7: Результаты бенчмарков сумма от 1 до 200 и чередующаяся сумма от 1 до 200.

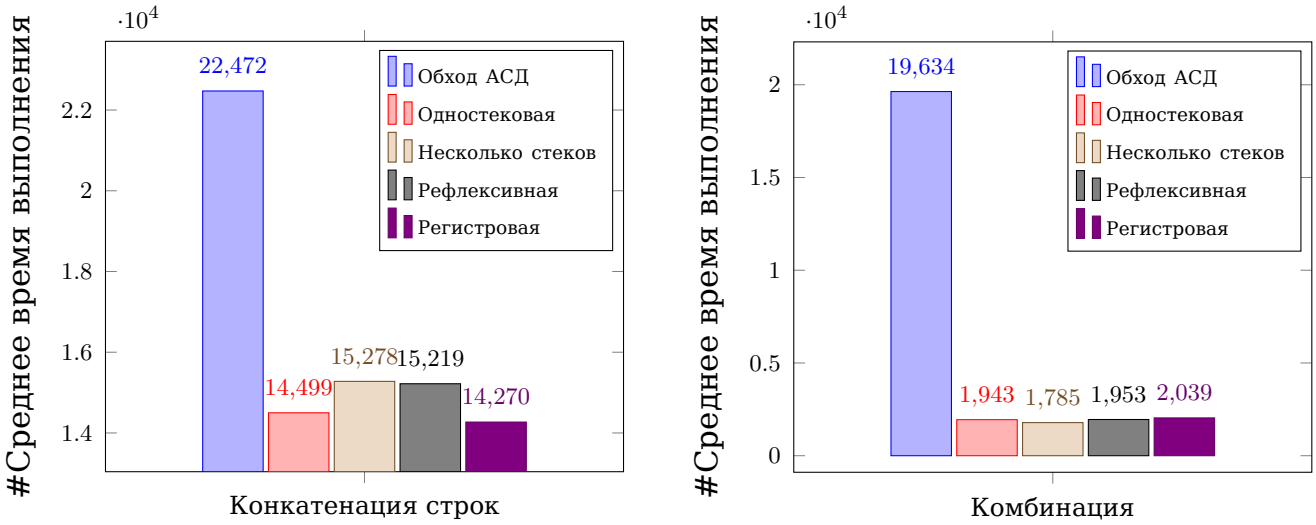

Рис. 8: Результаты бенчмарков конкатенация строк и комбинация сравнения строк и логических операций с булевыми значениями.

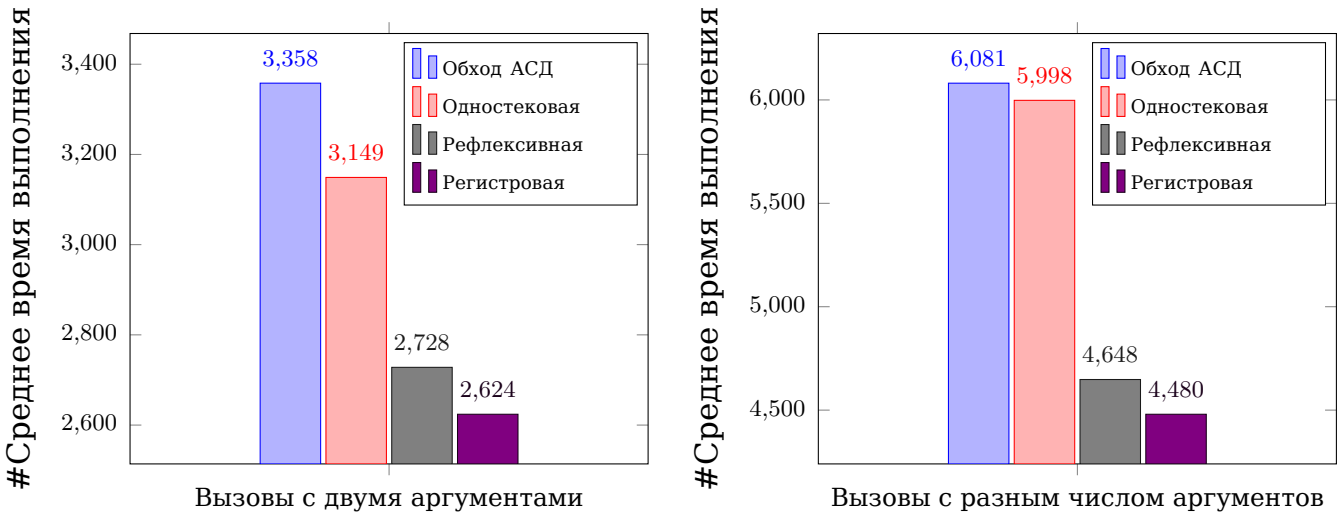

Рис. 9: Результаты бенчмарков вызовы функций с одинаковым числом аргументов и с разным числом аргументов.

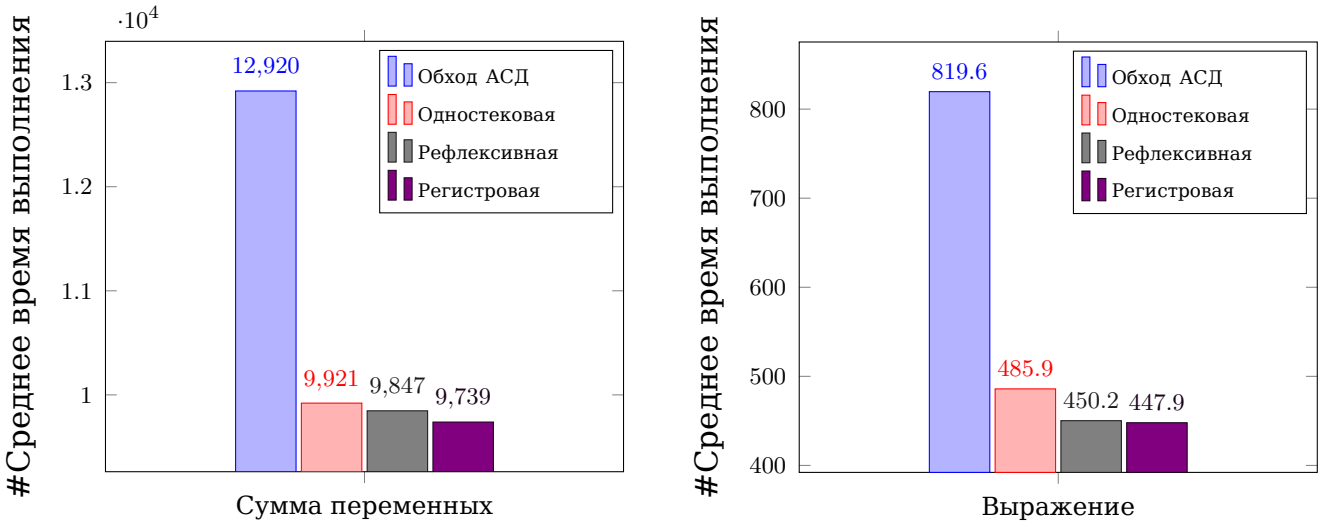

Рис. 10: Результаты бенчмарков сумма внешних переменных и выражение.

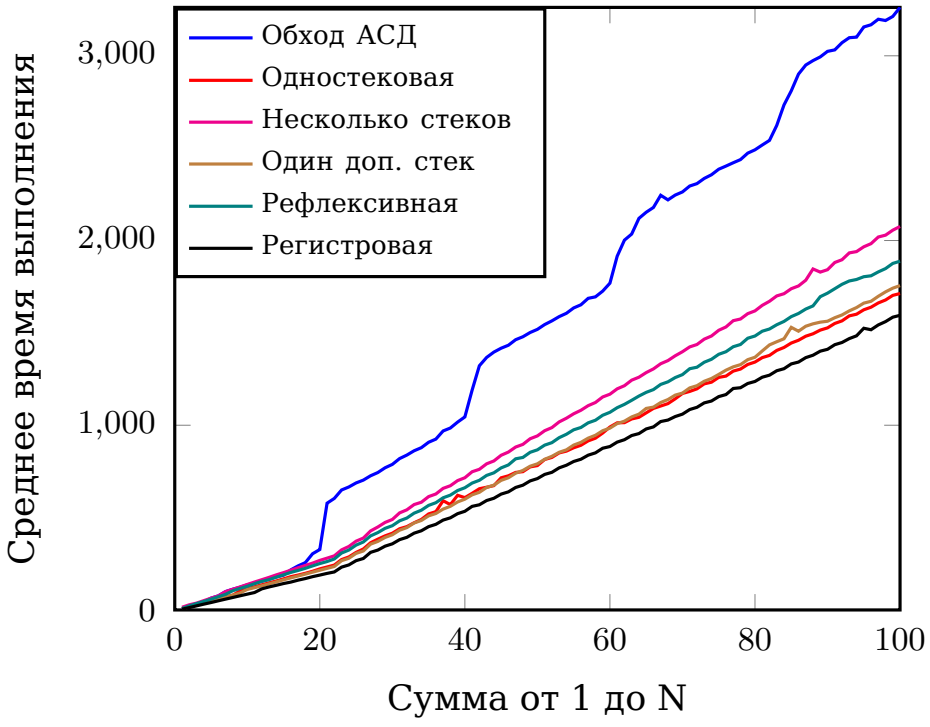

Рис. 11: Результаты бенчмарков для суммы чисел.

Анализируя данные на графиках, можно сделать следующие выводы:

- Виртуальная машина, использующая обход АСД, работает медленнее других виртуальных машин, так как она обрабатывает дерево напрямую, а не байт-код, что приводит к затратам по времени и памяти, особенно это заметно при работе с большими выражениями (в этом случае соответствующее абстрактное синтаксическое дерево достаточно объемное). Обход дерева требует посещения каждого узла и выполнения соответствующего кода. Каждый узел дерева содержит много информации, включая данные о поддеревьях, что приводит к разрозненным обращениям к памяти и дополнительным затратам на сборку мусора. В отличие от этого, стековые и регистровые машины работают с байт-кодом, и единственным негативным фактором, который замедляет скорость работы, является диспетчеризация с помощью оператора $switch$. Однако машина, использующая обход АСД, обладает преимуществом в простоте реализации.
- На графике с комбинацией сравнения строк и булевых операций видна ощутимая разница в скорости виртуальной машины, использующей АСД с остальными виртуальными машинами, поскольку стековые и регистровые виртуальные машины используют инструкции перехода (jump instructions) при реализации операторов OR и AND, что невозможно реализовать в виртуальной машине, использующей обход АСД.
- Виртуальная машина с несколькими стеками имеет сравнимую производительность при добавлении только одного стека (см. Рис. 11). Однако при добавлении нескольких стеков

(в нашем случае стека для строк и стека для целых чисел) возникают дополнительные затраты на определение какие данные из каких стеков использовать для выполнения нужных операций.

- При использовании рефлексивной машины можно заметить улучшение при вызове внешних функций, поскольку в сравнении с одностековой, которая хранит на стеке значения типа $interface\{\}$, не происходит дополнительных преобразований между $reflect.Value$ и $interface\{\}$. Однако скорость остальных операций ухудшается из-за доступа к данным через $reflect$, что медленее доступа через $interface\{\}$. В случае, если встраиваемый язык основан на вызове внешних функций, то данная виртуальная машина может быть решением, способным улучшить производительность.

- Регистровая виртуальная машина работает в среднем быстрее, чем одностековая, примерно на 25%, поскольку размер байт-кода меньше и требует явного указания адреса операндов. Стоит отметить, что наша реализация регистровой машины достаточно проста и добавление различных оптимизаций (например, оптимизация распределения регистров, сохранение несколько раз встречающихся значений/выражений в одном регистре и др.) может значительно улучшить скорость работы.

4.2 Тестирование

Для тестирования корректности работы были реализованы юнит-тесты, обеспечивающие покрытие на уровне 85%, для лек-

сера, парсера, компилятора для стековых виртуальных машин и для каждой виртуальной машины.

4.3 Выводы и результаты по главе

Для сравнения виртуальных машин использовались встроенные инструменты для написания бенчмарков из пакета $testing$. Было проведено сравнение виртуальных машин на 8 различных бенчмарках. Кроме того, были написаны юнит-тесты для тестирования корректности данной работы.

5 Заключение

В ходе выполнения данной работы были получены следующие результаты:

- Определен синтаксис и возможности встраиваемого языка программирования в Go.
- Для построения абстрактного синтаксического дерева были реализованы лексер и парсер для этого языка.
- Для генерации байт-кода для стековых виртуальных машин был реализован компилятор, для регистровых виртуальных машин были написаны соответствующие функции, генерирующие байт-код для определенных бенчмарков, используемых при сравнении виртуальных машин. Виртуальная машина, использующая обход АСД, не требует реализацию компилятора.

- Реализовано пять различных виртуальных машин: виртуальная машина с использованием обхода абстрактного синтаксического дерева, три стековые виртуальные машины (с одним стеком, с несколькими стеками и использующая рефлексию) и регистровая виртуальная машина.
- Написаны бенчмарки для сравнения производительности виртуальных машин и проведено их сравнение, которое показало, что:
  - Виртуальная машина, использующая обход АСД работает гораздо медленнее виртуальных машин, работающих с байт-кодом.
  - Использование нескольких стеков не ускоряет работу одностековой виртуальной машины. При вызове функций рефлексивная машина работает гораздо лучше, чем одностековая виртуальная машина, использующая $interface\{\}$. Однако при выполнении других операций скорость выполнения ухудшается.
  - Регистровая виртуальная машина в среднем тратит примерно на 25% меньше времени, чем стековая.
- Для тестирования корректности работы были написаны юнит-тесты с 85% покрытием.

Результаты, полученные в данной работе, можно использовать для выбора виртуальной машины при реализации встраиваемого языка в Go, а также улучшения производительности встраиваемых языков программирования.

Дальнейшая работа возможна в следующих направлениях:

- Исследование других виртуальных машин для встраиваемого языка в Go с учетом различных особенностей языка. Например, можно модифицировать байт-код виртуальной машины с несколькими стеками, добавив определенные опкоды для добавления и удаления со стека для соответствующих типов данных. Такой подход позволит сократить количество проверок на то, с какими данными из каких стеков мы должны выполнять определенные операции.
- Реализация комбинированной виртуальной машины, которая может использовать разные подходы в зависимости от типа выражений. Например, для вызова функций может быть использована рефлексивная виртуальная машина, а для других выражений - стековая.

Список литературы

1. *Aycock J.* A brief history of just-in-time. // ACM Computing Surveys. — 2003.
2. *Ball T.* Writing a Compiler in Go. — 2019.
3. *Ball T.* Writing an Interpreter in Go. — 2019.
4. *Bell J. F.* Threaded code // Communications of the ACM. — 1973.
5. *Chizhova M.* Bachelor thesis. — 2023. — URL: `https://github.com/MariaChizhova/bachelor-thesis/`.
6. Compilers: Principles, Techniques, and Tools. / A. V. Aho [и др.]. — Addison-Wesley / Helix Books, 2013.
7. *Dewar R.* Indirect threaded code // Communications of the ACM. — 1975.
8. *Ertl M. A., Gregg D.* The structure and performance of Efficient interpreters // The Journal of Instruction-Level Parallelism. — 2003.
9. *Fang R., Liu S.* A Performance Survey on Stack-based and Register-based Virtual. — 2016.
10. *Google*. A fast, scalable, multi-language and extensible build system. — 2023. — URL: `https://bazel.build/`.
11. *Google*. Common Expression Language. — 2023. — URL: `https://github.com/google/cel-go`.
12. *Google*. Starlark in Go: the Starlark configuration language, implemented in Go. — 2022. — URL: `https://github.com/google/starlark-go`.

13. *Grune D., Jacobs C. J.* Parsing Techniques: A Practical Guide. // Springer Science. — 2007.
14. *Hashicorp*. Generic boolean expression evaluation in Go. — 2023. — URL: https://github.com/hashicorp/go-bexp.
15. *Ierusalimschy R., Celes W., De Figueiredo L. H.* The Implementation of Lua 5.0. — 2005.
16. *Kemp S.* An embeddable evaluation-engine. — 2022. — URL: https://github.com/skx/evalfilter.
17. *Krimen R.* A JavaScript interpreter in Go. — 2022. — URL: https://github.com/robertkrimen/otto.
18. *Lester G.* Arbitrary expression evaluation for golang. — 2017. — URL: https://github.com/Knetic/govaluate.
19. *Medvedev A. A.* Expression language and expression evaluation for Go. — 2023. — URL: https://github.com/antonmedv/expr.
20. *Medvedev A. A.* Go expression evaluation comparison. — 2023. — URL: https://github.com/antonmedv/golang-expression-evaluation-comparison.
21. *Norvell T.* Parsing Expressions by Recursive Descent. — 1999. — URL: https://www.engr.mun.ca/~theo/Misc/exp_parsing.htm.
22. *Nystrom R.* Crafting Interpreters. — 2021.
23. *PaesslerAG*. Expression evaluation in golang. — 2023. — URL: https://github.com/PaesslerAG/gval.
24. *Panov D.* ECMAScript/JavaScript engine in pure Go. — 2023. — URL: https://github.com/dop251/goja.
25. *Parr T.* ANTLR. — 2023. — URL: http://www.antlr.org.

26. *Parr T.* Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages. — 2009.
27. *Pike R.* Lexical Scanning in Go. — 2011. — URL: https://go.dev/talks/2011/lex.slide#1.
28. *Shopify*. A Lua VM in Go. — 2016. — URL: https://github.com/Shopify/go-lua/blob/88a6f168eee0ba102d7d20c5281056a5dd3d7vm.go#L306.
29. *The Go Authors*. cmd/compile: compile dense, pure-integer “switch” into jump table. — 2013. — URL: https://github.com/golang/go/issues/5496.
30. *The Go Authors*. cmd/compile: improve escape analysis of make([]T, n) where n is non-constant. — 2017. — URL: https://github.com/golang/go/issues/20533.
31. *The Go Authors*. The Go Programming Language. — 2023. — URL: https://go.dev.
32. *The Protocol Buffers Authors*. Protocol Buffers Documentation. — 2023. — URL: https://protobuf.dev/.
33. *The Starlark Authors*. Starlark in Go: Implementation. — 2021. — URL: https://github.com/google/starlark-go/blob/master/doc/impl.md#evaluator.
34. *Van Gijzel B.* Comparing Parser Construction Techniques. — 2009.
35. Virtual Machine Showdown: Stack Versus Registers / Y. Shi [и др.] // ACM Transactions on Architecture and Code Optimization. — 2008.