Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ Информатика и системы управления

КАФЕДРА Программное обеспечение ЭВМ и информационные технологии

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
К КУРСОВОЙ РАБОТЕ
НА ТЕМУ:

Компилятор языка программирования Golang

Студент группы ИУ7-22М _____ В.Н. Ларин
(Подпись, дата) (И.О. Фамилия)

Руководитель _____ А.А. Ступников
(Подпись, дата) (И.О. Фамилия)

2024 г.

РЕФЕРАТ

Отчет содержит 34 стр., 9 рис., 1 табл., 18 источн., 2 прил.

Ключевые слова: компиляция, сбор мусора, Go, Golang, LLVM, AST, ANTLR, исполняемый код, анализатор, строитель дерева.

Предмет исследования – компилятор. Объект исследования –построение компилятора языка GoLang.

Компилятор – это программное средство, которое преобразует исходный текст программы на определенном языке программирования в машинный код для исполнения на целевой архитектуре. Golang – это многопоточный язык программирования, разработанный Google для создания распределенных систем. Был проведен анализ инфраструктуры разработки компилятора Golang, включающий сравнительное изучение инструментов для генерации лексических и синтаксических анализаторов, а также аспектов типизации и управления ресурсами. Исследование привело к разработке концептуальной модели компилятора, методов обработки пакетов, системы представления типов и структуры построения промежуточного представления кода. Для реализации проекта были выбраны LLVM, ANTLR4, язык программирования C++. Разработанные программы для тестирования компилятора охватывают разнообразные языковые конструкции.

Полученные результаты являются основой для будущей работы над развитием и оптимизацией компилятора Golang. Было проведено тестирование функциональности компилятора и сборщика мусора.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Компилятор – программное обеспечение, переводящее текст программы, написанный на определенном языке программирования (исходном), в машинный код, который может быть исполнен на вычислительном устройстве. Процесс компиляции включает в себя оптимизацию кода и анализ ошибок, что способствует повышению производительности и предотвращению некоторых сбоев во время выполнения программы. Компилятор должен быть способен обрабатывать текстовые файлы, содержащие исходный код, и создавать программу, готовую к выполнению [1].

Golang – компилируемый многопоточный язык программирования, разработанный внутри компании Google. Разработка Go началась в сентябре 2007 года, его непосредственным проектированием занимались Роберт Гризмер, Роб Пайк и Кен Томпсон. Официально язык был представлен в ноябре 2009 года. Go поддерживается набором компиляторов gcc, существует несколько независимых реализаций. Язык Go разрабатывался как язык программирования для создания высокоэффективных программ, работающих на современных распределённых системах и многоядерных процессорах [2].

Целью работы является разработка компилятора для языка программирования Golang. Для достижения поставленной цели необходимо решить следующие задачи:

— проанализировать грамматику языка Golang;

— изучить существующие подходы к анализу исходных кодов программ, а также системы для генерации низкоуровневого кода;

— разработать прототип компилятора, способного преобразовывать исходный код на языке Golang в машинный код.

1 Аналитический раздел

1.1 Инфраструктура построения компилятора

Инфраструктура построения компилятора обычно состоит из трех основных компонентов: Frontend, Middle-end и Backend. На рисунке 1.1 представлена схема слоев инфраструктуры компилятора.

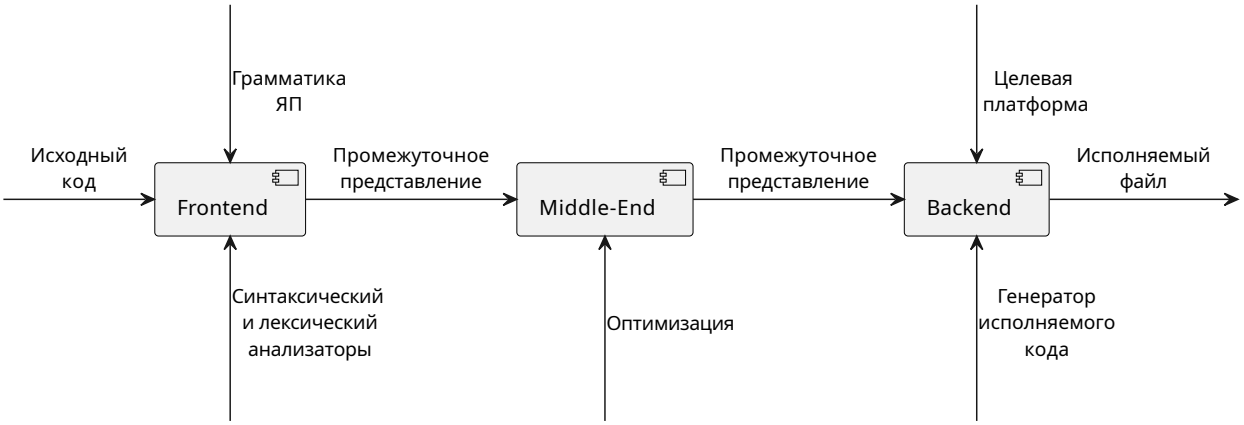

Рисунок 1.1 — Слои компилятора

Frontend компилятора отвечает за анализ исходного кода программы. Данный компонент обычно включает в себя препроцессор, лексические, синтаксические и семантические анализаторы, генераторы промежуточного представления.

Middle-end выполняет оптимизации и преобразования программы, основанные на промежуточном представлении, полученном от Frontend. Данный компонент может включать в себя различные оптимизации, такие как удаление недостижимого кода, упрощение выражений, встраивание функций. Middle-end также может выполнять анализ зависимостей, определение времени жизни переменных и иные анализы, необходимые для оптимизации программы. После завершения работы Middle-end создает промежуточное представление программы, которое будет передано Backend для генерации машинного кода.

Backend компилятора отвечает за генерацию машинного кода на основе промежуточного представления программы, полученного от Middle-end. Данный компонент анализирует промежуточное представление и генерирует соответствующий машинный код для целевой архитектуры. Backend также может выполнять дополнительные оптимизации, специфичные для целевой архитектуры, такие как распределение регистров и сокращение размера кода. После завершения работы Backend генерирует исполняемый файл, который может быть запущен на целевой аппаратной платформе [3].

1.1.1 Лексические и синтаксические анализаторы

Лексический и синтаксический анализаторы являются ключевыми компонентами компилятора, ответственными за преобразование исходного кода программы в промежуточное представление, которое необходимо для дальнейшей обработки.

Лексический анализатор (лексер) преобразует поток символов исходного кода программы в последовательность токенов. Токены представляют собой лексемы, или элементы языка, такие как идентификаторы, ключевые слова, операторы, числа и строки. Лексический анализатор также может игнорировать комментарии и обрабатывать различные форматирования, такие как пробелы и переводы строк.

Синтаксический анализатор (парсер) анализирует структуру программы на основе последовательности токенов, созданных лексическим анализатором. Он использует грамматику языка программирования для определения правил, по которым могут комбинироваться токены, чтобы создавать корректные конструкции языка. Синтаксический анализатор создает синтаксическое дерево (дерево разбора), которое представляет собой иерархическое представление структуры программы [4, 5].

Существует несколько подходов для реализации лексического и синтаксического анализаторов:

— с использованием стандартных алгоритмов анализа;

— с привлечением готовых инструментов генерации.

Методы лексического анализа

LL-анализатор – это метод разбора для определенного подмножества контекстно-свободных грамматик, которые известны как LL-грамматики. Буква L в термине «LL-анализатор» обозначает, что входная строка обрабатывается с начала до конца и генерируется ее левосторонний вывод. Если анализатор использует предварительный просмотр на k токенов при разборе входных данных, его называют LL(k)-анализатор. LL(1)-анализаторы очень широко распространены, так как они просматривают входные данные только на один шаг вперед, чтобы определить, какое грамматическое правило применить.

LR-анализатор читает входной поток слева направо и создает самую правую продукцию контекстно-свободной грамматики. Термин LR(k)-анализатор также используется. Здесь k означает количество непрочитанных символов предварительного просмотра во входном потоке, на основе которых принимаются решения в процессе анализа. Синтаксис многих языков программирования определяется грамматикой LR(1) или аналогичной грамматикой. LR-анализатор обрабатывает код сверху вниз, поскольку он пытается создать продукцию верхнего уровня грамматики из листьев [6].

Генераторы лексических и синтаксических анализаторов

Lex [7] – стандартный инструмент для получения лексических анализаторов в операционных системах Unix, обычно используется совместно с генератором синтаксических анализаторов Yacc. В результате обработки входного потока получается исходный файл на языке Си. Flex (Fast Lexical Analyzer) [8] заменяет Lex в системах на базе пакетов GNU и имеет аналогичную функциональность. Bison – аналогичный YACC генератор для GNU систем.

ANTLR (ANother Tool for Language Recognition) – генератор лексических и синтаксических анализаторов, позволяет создавать анализаторы на многих языках программирования. Он также позволяет строить и обходить деревья синтаксического анализа с использованием паттернов посетитель или слушатель [9].

Coco/R – инструмент генерации компиляторов или интерпретаторов языка. Coco/R изначально разрабатывался в ETHZ и перешел вместе с Ханспетером Мессенбеком в Университет Линца. Coco/R распространяется на условиях слегка смягченной GNU General Public License [10].

Сравнительный анализ генераторов

В таблице 1.1 представлен сравнительный анализ анализаторов исходного кода.

Таблица 1.1 — Сравнение генераторов анализаторов исходного кода

Критерии	ANTLR	Flex + Bison	Coco/R
Алгоритм лексера	ДКА	ДКА	ДКА
Алгоритм парсера	LL(*)	LR	LL(1)
Поддерживаемые языки программирования	C#, Java, Python, JavaScript, C++, Swift, Go, PHP	C, C++, Java	C, C++, C#, F#, Java, Ada, Object Pascal, Delphi, Modula-2, Oberon, Ruby, Swift, Unicon, Visual Basic
Входная грамматика	РБНФ	Yacc	РБНФ
Интеграция с инструментами разработки	IntelliJ IDEA, Eclipse и другие	Нет	Нет
Наличие встроенных инструментов отладки	Да	Нет	Нет
Лицензия использования	BSD-подобная	GNU GPL	GNU GPL

Все три инструмента предлагают различные подходы к синтаксическому анализу текста. Данные инструменты могут работать с РБНФ грамматиками. ANTLR имеет преимущество в виде поддержки интегрированных инструментов разработки, таких как IntelliJ IDEA и Eclipse. C

точки зрения лицензии, ANTLR является одним из наиболее гибких инструментов, предоставляя BSD-подобную лицензию на использование.

1.1.2 Генерация исполняемого кода

Инструменты генерации исполняемого кода играют ключевую роль в процессе компиляции программ. Они отвечают за трансляцию абстрактного синтаксического дерева (AST-дерева) или промежуточного представления (IR) программы в исполняемый машинный код или байткод, который может быть исполнен на целевой аппаратуре или виртуальной машине.

Генерация машинного кода относится к процессу преобразования исходного кода программы в низкоуровневый бинарный код, который может быть исполнен непосредственно аппаратурой процессора. Генерация машинного кода приводит к созданию исполняемых файлов, которые могут быть непосредственно запущены на целевой платформе без дополнительной обработки.

Виртуальная машина – это абстрактная вычислительная машина, которая исполняет программный код, представленный в виде некоторого промежуточного представления – байткода. Виртуальные машины обычно используются для исполнения программного кода, который был предварительно скомпилирован в промежуточное представление, а не в машинный код конкретной архитектуры процессора [11]. Примерами виртуальных машин являются Java Virtual Machine (JVM), Common Language Runtime (CLR), и V8 (WASM) для веб-приложений .

LLVM (Low Level Virtual Machine) представляет собой инфраструктуру для разработки компиляторов и связанных инструментов. Он включает в себя мощный оптимизатор и генератор машинного кода, который может работать с различными архитектурами процессоров. LLVM генерирует промежуточное представление LLVM IR, которое затем транслируется в машинный код для целевой архитектуры. Он также поддерживает генерацию байткода для виртуальных машин, таких как JVM и WASM [12].

GCC (GNU Compiler Collection) - это другая широко используемая инфраструктура для разработки компиляторов. В качестве промежуточного представления используется GIMPLE. GCC генерирует машинный код для различных архитектур процессоров и поддерживает генерацию байткода для некоторых виртуальных машин [13].

Java ASM (Java bytecode manipulation framework) - это библиотека на языке Java, предназначенная для манипулирования байткодом Java. Ее можно использовать для модификации существующих классов или для динамической генерации классов непосредственно в форме байткода. ASM предоставляет некоторые распространенные преобразования байт-кода и алгоритмы анализа, на основе которых могут быть созданы пользовательские сложные преобразования и инструменты анализа кода. ASM предлагает функциональность, аналогичную другим фреймворкам байт-кода Java, но ориентирован на производительность.

ILAsm (Intermediate Language Assembler) – это инструмент для создания исполняемых программ для платформы CLR (Common Language Runtime), использующей байткод Intermediate Language (IL). Он позволяет разработчикам создавать IL код вручную и компилировать его в исполняемый байткод с помощью утилиты компиляции IL.

WASM – это низкоуровневый бинарный формат, предназначенный для исполнения в веб-браузерах. Binaryen – это инструментарий для манипулирования и оптимизации байткода WebAssembly. Он обеспечивает высокую производительность и безопасность и может быть использован для исполнения приложений на различных языках программирования в веб-среде.

В данной работе использование инструмента LLVM для генерации кода на целевую платформу x64 было обусловлено его мощными возможностями и гибкостью. LLVM предоставляет инфраструктуру для разработки компиляторов и связанных инструментов, включая оптимизатор и генератор машинного кода, способный работать с различными архитектурами процессоров.

1.2 Особенности языка Golang

Программа на Go состоит из пакетов, которые могут быть использованы для организации кода. Каждый файл программы должен начинаться с объявления пакета. Подключение пакетов выполняются с помощью ключевого слова import.

Переменные объявляются с использованием ключевого слова var, за которым следует имя переменной и ее тип, либо с помощью оператора краткого объявления :=.

Функции объявляются с использованием ключевого слова func, за которым следует имя функции, список параметров и тип возвращаемого значения (если оно есть). Функции также могут возвращать несколько значений.

Golang поддерживает стандартные управляющие конструкции, такие как условные операторы if, циклы for, switch и операторы break, continue для управления выполнением программы.

Go включает в себя встроенную поддержку конкурентности с помощью горутин и каналов. Горутины представляют собой легковесные потоки выполнения, а каналы - механизм для обмена данными между горутинами.

Типизация

В статически типизированных языках типы данных определяются на этапе компиляции. Переменные должны быть объявлены с указанием их типа, и этот тип не может быть изменен во время выполнения программы. Проверка типов происходит на этапе компиляции, что позволяет выявлять множество ошибок ещё до запуска программы.

В динамически типизированных языках типы данных определяются во время выполнения программы. Переменные могут содержать значения разных типов, и их тип может изменяться во

время выполнения программы. Проверка типов происходит во время выполнения программы, что может приводить к ошибкам во время выполнения, если типы несовместимы.

Статическая типизация обычно обеспечивает более строгую проверку типов, что помогает выявлять ошибки на этапе компиляции, но может потребовать больше времени и усилий от разработчика для объявления и управления типами. Динамическая типизация обычно обеспечивает более гибкую и удобную работу с типами данных, но может привести к неожиданным ошибкам во время выполнения, особенно в больших и сложных программах.

В языке программирования Golang используется статическая типизация. В Go типы данных обычно указываются явно при объявлении переменных, функций и других элементов программы. В Go также поддерживается краткое объявление переменных с использованием оператора **:=**, который автоматически выводит тип переменной на основе значения, с которым она инициализируется.

Go предоставляет разнообразие встроенных типов данных, включая числовые типы (int, float, complex), строковые типы (string), булев тип (bool), указатели (*T), массивы ([n]T), срезы ([]T), карты (map[K]V), структуры (struct) и интерфейсы (interface). В Go можно создавать пользовательские типы данных с помощью ключевого слова type. Это позволяет абстрагировать сложные структуры данных и создавать собственные абстракции. В Go у каждого типа данных есть свое нулевое значение. Например, для числовых типов это 0, для строк это пустая строка, а для указателей это nil.

Интерфейсы в Go определяют набор методов, которые должен реализовать тип данных, чтобы удовлетворить интерфейс. Это позволяет использовать полиморфизм в коде и обеспечивает гибкость при разработке

Работа с ресурсами

Golang использует механизм сбора мусора для автоматического управления памятью. Сборщик мусора следит за использованием объектов и автоматически освобождает память, занимаемую объектами, которые больше не используются. В данном языке программирования освобождение неиспользуемой памяти запускается автоматически по мере необходимости, когда система обнаруживает, что памяти становится недостаточно или размер кучи увеличился в два раза. Такой подход позволяет избежать неэффективного использования ресурсов и уменьшает накладные расходы на управление памятью.

В Go можно использовать указатели для явного управления памятью. Go предоставляет безопасные и удобные абстракции для работы с памятью, что уменьшает риск ошибок и повышает производительность кода.

В Go предусмотрен механизм отложенного выполнения **defer**, который позволяет освобождать ресурсы (например, файлы или сетевые соединения) после завершения работы с ними.

Это помогает избежать утечек ресурсов и обеспечивает эффективное использование системных ресурсов.

Пакеты

В языке программирования Go пакеты играют важную роль в организации кода и обеспечении его модульности и повторного использования. Пакет в Go - это набор связанных между собой файлов с исходным кодом, находящихся в одном каталоге.

Существуют два типа пакетов в Go: исполняемые и библиотечные. Исполняемый пакет представляет собой программу, которая может быть запущена. Он содержит функцию **main**, с которой начинается выполнение программы. Библиотечный пакет представляет собой библиотеку кода, предназначенную для повторного использования в других программах.

Для использования функций, типов и переменных из других пакетов необходимо импортировать их в текущий файл с помощью ключевого слова **import**. При импортировании пакета в Go выполняется его инициализация. Инициализация выполняется перед выполнением функции ‘main‘ в случае исполняемого пакета.

В Go пакеты организованы в виде древовидной структуры. Имена пакетов должны быть уникальными в пределах пространства имен. Подкаталоги внутри каталога пакета могут содержать дополнительные пакеты.

В Go принята концепция видимости символов. Идентификаторы (функции, типы, переменные) считаются видимыми только внутри пакета, если они начинаются с заглавной буквы. Идентификаторы, начинающиеся с прописной буквы, не доступны извне пакета.

Go поддерживает использование сторонних пакетов, которые не входят в стандартную библиотеку. Данные пакеты обычно устанавливаются с помощью менеджера зависимостей ‘go mod‘.

1.3 Управление памятью

Управление памятью (как автоматическое, так и ручное) – это процесс управления памятью. В частности, существуют две особые проблемы, связанные с управлением памятью:

а) предоставлять память по запросу (выделение памяти);

б) освобождать неиспользуемую память для повторного использования в будущем (освобождение памяти).

Данная проблема является классической в любой операционной системе. Управление памятью также является фундаментальной проблемой для языков программирования, поскольку они должны автоматически управлять памятью [14].

Задача сбора мусора

Задача управления памятью в Golang обрабатывается сборщиком мусора (GC), который отвечает за несколько задач:

а) выделять память для новых объектов

б) следить за тем, чтобы все используемые объекты хранились в памяти

в) очищать память, которая больше не используется (мусор)

Сборщиком мусора – это набор алгоритмов, которые скрывают большинство проблем с управлением памятью от языков высокого уровня. Однако важно отметить, что GC не решает всех проблем с памятью [15]. Например, ничто не мешает приложению:

а) сохранять ссылки на данные, которые больше никогда не будет использоваться;

б) выделять объекты на неопределенный срок (и сохранять ссылки на все из них) до тех пор, пока не закончится память.

В таких сценариях нет недоступных объектов, и, следовательно, GC не может освободить какую-либо память, что приводит к ошибке нехватки памяти, которая в конечном итоге завершает работу приложения.

Сборщики мусора обладают следующими характеристиками [16].

— Безопасность: сборщик не должен освобождать используемые объекты.

— Полнота – это характеристика показывающая, что весь мусор в конечном итоге будет очищен.

— Оперативность – промежуток времени между появлением мусором и принятием решения о его удалении.

— Время паузы – время необходимое сборщику для определения мусора и его утилизации.

— Накладные расходы на пространство – объем пространства, необходимый для работы сборщика и сохранения дополнительной информации об объектах.

— Масштабируемость – возможность к оперированию большим количеством объектов без снижения производительности приложения.

Алгоритмы подсчета ссылок

Как следует из названия, алгоритмы подсчета ссылок (впервые представленные в работе [17]) буквально подсчитывают ссылки на объекты. Такие алгоритмы основаны на следующем инварианте: объект считается живым тогда и только тогда, когда количество ссылок на объект больше нуля. Следовательно, чтобы иметь возможность узнать, является ли объект живым или

нет, алгоритмы подсчета ссылок сохраняют счетчик ссылок для каждого объекта. Подсчет ссылок считается прямым GC, поскольку он идентифицирует мусор, то есть объекты без входящих ссылок.

Алгоритмы трассировки ссылок

Алгоритмы трассировки ссылок основаны на обходе графа объектов и маркировке достижимых объектов. Трассировка ссылок довольно проста; объекты, которые отмечены во время трассировки ссылок, считаются живыми. Все позиции памяти, которые не отмечены, считаются мусором и будут освобождены. Следовательно, трассировка ссылок считается косвенной GC, то есть она обнаруживает не мусор, а живые объекты [18].

Данных подход также называется трехцветной маркировкой, где каждый объект может находиться в одном из следующих состояний:

— белый – начальное состояние;

— черный – у объекта нет исходящих ссылок на белые объекты;

— серый – у объекта есть ссылки на белые объекты.

Трехцветная разметка начинается с помещения всех корневых объектов в серый набор, а всех остальных объектов - в белый набор. Затем алгоритм выполняется следующим образом:

— пока в сером наборе есть объекты,

  — выбрать один объект из серого набора,

  — переместить его в черный набор,

  — поместить все объекты, на которые он ссылается, в серый набор.

1.4 Формальная постановка задачи

Необходимо разработать прототип компилятора языка программирования Golang. Для генерации парсеров и лексеров грамматики языка необходимо использовать ANTLR4. Для построения IR необходимо использовать библиотеку LLVM. Для компиляции IR для целевой архитектуры x64 также необходимо использовать LLVM. На рисунке 1.2 представлена IDEF0 диаграмма компилятора языка программирования Golang.

Выводы

В ходе исследования был рассмотрен процесс построения компилятора, включая анализ инфраструктуры и сравнительный анализ инструментов для генерации лексических и синтаксических анализаторов. Были выявлены особенности типизации, управления ресурсами и организации

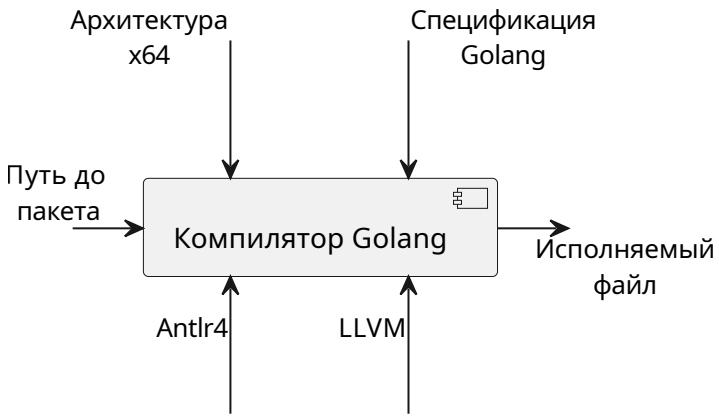

Рисунок 1.2 — IDEF0 диаграмма компилятора языка программирования Golang

пакетов в языке программирования Golang. Также был проведен обзор области управления памятью в контексте разработки компилятора. В итоге была сформулирована формальная задача разработки компилятора для языка Golang.

2 Конструкторский раздел

2.1 Концептуальная модель

На рисунке 2.1 представлен схема процесса компиляции языка программирования Golang.

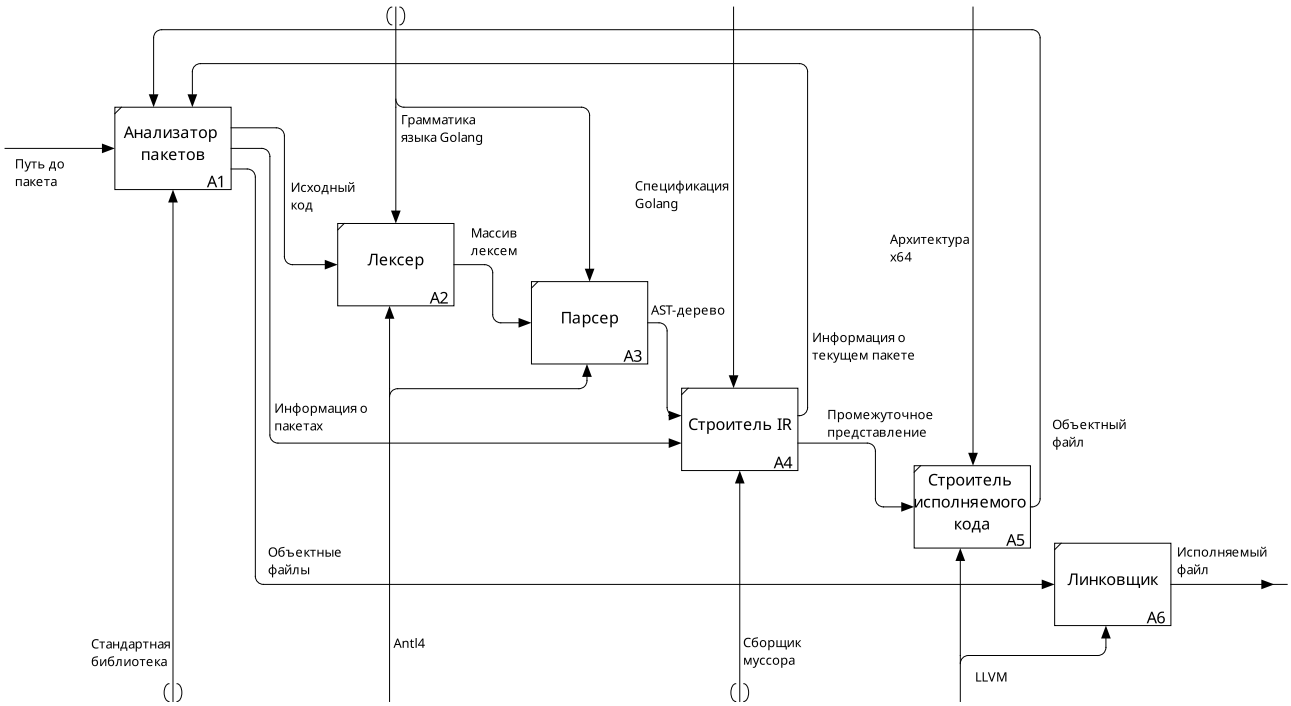

Рисунок 2.1 — Детализированная IDEF0 диаграмма компилятора языка программирования Golang

Данная диаграмма разделяет компилятор на ряд компонентов. Анализатор пакетов принимает путь до пакета и использует стандартную библиотеку, передавая далее исходный код. Лексер трансформирует исходный код в массив лексем, а парсер на основе данного массива строит AST-дерево. Строитель промежуточного представления IR использует AST-дерево и информацию о пакетах для создания промежуточного представления. Затем строитель исполняемого кода конвертирует промежуточное представление в объектный файл с помощью LLVM. Линковщик собирает объектные файлы и формирует исполняемый файл.

2.2 Система пакетов

Анализатор пакетов строит дерево зависимостей и поочередно от листов к корню собирает пакеты. Корневым пакетом является исполняемый пакет, имеет имя **main** и функцию **main**.

Представление пакета

Было разработано представление пакета и базовых типов, представленное на рисунке 2.2. Пакет содержит имя пакета, его хэш-имя, которое будет использоваться для декларации функций

и констант в объектных файлах для их линковки. Представление также содержит хэш-таблицы на объявленные типы, функции, константы.

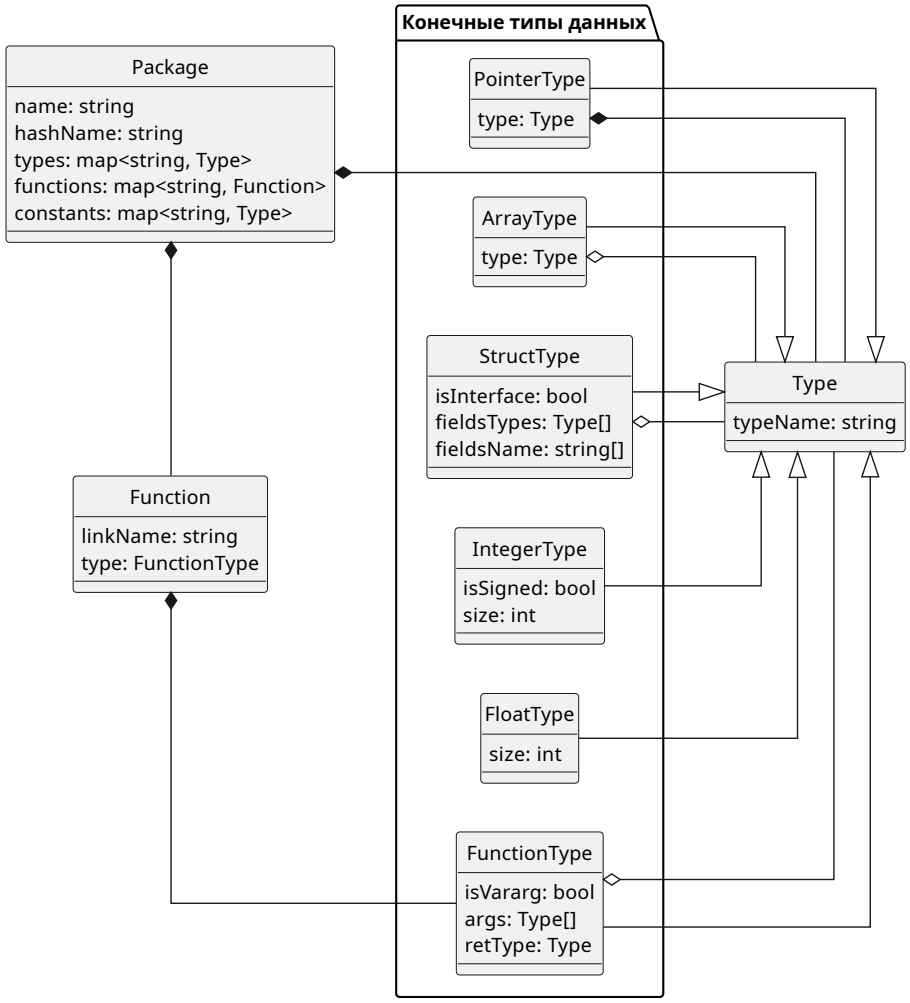

Рисунок 2.2 — Представление пакета

Была предложена поддержка следующих базовых типов. IntegerType – целое число длины size бит, обладающее знаком согласно флагу isSigned. FloatType – число с плавающей запятой длины size бит. StructType – структура, содержащая поля с именами FieldNames и типами FieldsTypes. ArrayType – массив, содержащий элементы типа type. PointerType – указатель на тип type. FunctionType – функция, имеющая аргументы типы args и возвращаемый тип retType.

Данного представления достаточно, чтобы поддерживать базовые типы языка: bool, byte, float32, float64, int, int16, int32, int64, int8, rune, string, uint, uint16, uint32, uint64, uint8, – создавать пользовательские структуры, интерфейсы, функции, методы и экспортировать их из пакета Пример пакета стандартной библиотеки builtin представлен в приложении А.

Дерево зависимостей пакетов

Алгоритм 2.1 реализует построение дерева зависимостей для пакетов. Он начинает с корневого пакета и использует обход графа зависимостей для добавления зависимостей в дерево.

Каждая зависимость обрабатывается только один раз, что позволяет избежать циклических зависимостей и гарантирует эффективное построение дерева.

Алгоритм 2.1 — Формирования дерева зависимостей

1: **Функция** $BuildDependencyTree(\ rootPackage)$
▷ $rootPackage$ — корней пакет

2: $dependencyTree \leftarrow$ создать граф
3: $queue \leftarrow$ создать очередь
4: добавить $rootPackage$ в очередь $queue$
5: **Пока** очередь $queue$ не пуста **выполнить**
6: $currentPackage \leftarrow$ взять из очереди $queue$
7: $dependencies \leftarrow$ получить массив зависимостей пакета $currentPackage$
8: **Для всех** $dependency \in dependencies$ **выполнить**
9: добавить ребро $currentPackage \rightarrow dependency$ в граф $dependencyTree$
10: **Если** $dependency \notin dependencyTree$ **тогда**
11: добавить $dependency$ в очередь $queue$
12: **Конец условия**
13: **Конец цикла**
14: **Конец цикла**
15: **Вернуть** $dependencyTree$
16: **Конец функции**

2.3 Построение анализаторов исходного кода

В данной работе лексический и синтаксический анализаторы создаются при помощи ANTLR. На вход принимается грамматика языка программирования в формате ANTLR4. Грамматика приведена в приложении Б. Она состоит из объявления ключевых слов и правил вывода.

В результате работы ANTLR4 генерируются файлы, содержащие классы лексического и синтаксического анализаторов, а также необходимые вспомогательные файлы и классы для их функционирования. Кроме того, создаются шаблоны классов для обхода дерева разбора, полученного в результате работы парсера.

Лексер принимает на вход текст программы, преобразованный в поток символов, и выдает поток токенов, который передается на вход парсера. После выполнения парсера создается дерево разбора. Любые ошибки, возникающие в процессе работы лексера и парсера, выводятся в стандартный поток ввода-вывода.

Абстрактное синтаксическое дерево (AST) предоставляет структурированное представление программы. Его можно обойти двумя основными способами: с помощью паттерна Listener и паттерна Visitor.

Паттерн Listener позволяет обходить дерево в глубину, вызывая обработчики соответствующих событий при входе и выходе из узла дерева. Когда происходит обход дерева, при входе в узел вызывается соответствующий обработчик, а при выходе - другой обработчик. Это позволяет реализовать обработку узлов на основе их типов и состояний.

Паттерн Visitor предоставляет более гибкий способ обхода дерева. Он позволяет определить, какие узлы и в каком порядке нужно посетить. Для каждого типа узла в дереве реализуется метод его посещения. Обход начинается с точки входа в программу, обычно с корневого узла. Каждый узел посещается в заданном порядке, что позволяет реализовать различные алгоритмы обработки AST-дерева.

2.4 Построение промежуточного представления (IR)

В данной работе был использован паттерн Visitor для обхода абстрактного синтаксического дерева. Данный подход обеспечивает гибкость и позволяет точно контролировать порядок посещения узлов дерева, что удобно при реализации различных анализов и преобразований.

На рисунке 2.3 представлена диаграмма классов, определяющая структуру строителей и посетителей для компилятора языка программирования Golang.

В ней есть два основных строителя: LlvmBuilder, который отвечает за создание LLVM IR кода, и GoIrBuilder, который является адаптером для LlvmBuilder и предоставляет интерфейс для создания промежуточного представления кода Golang. GoIrBuilder обладает достаточным интерфейсом для разработки компилятора данного языка программирования. Класс GoIrVisitor представляет собой посетителя, который обходит абстрактное синтаксическое дерево (класс AST) и выполняет различные операции над его узлами. Он расширяет базовый посетитель BaseVisitor, чтобы обеспечить специфическую функциональность для языка Golang. Связь между GoIrVisitor и GoIrBuilder показывает, что посетитель использует строитель для создания кода внутри методов обхода AST-дерева.

2.5 Алгоритмы управления памятью

Формирование теневого стека

Для отслеживания достижимых ссылок из кода программы предлагается использовать теневой стек. При вызове подпрограмм предлагается добавлять новый слой в него, а при возврате управления – удалять данный слой. В слое хранятся ссылки на выделенные объекты.

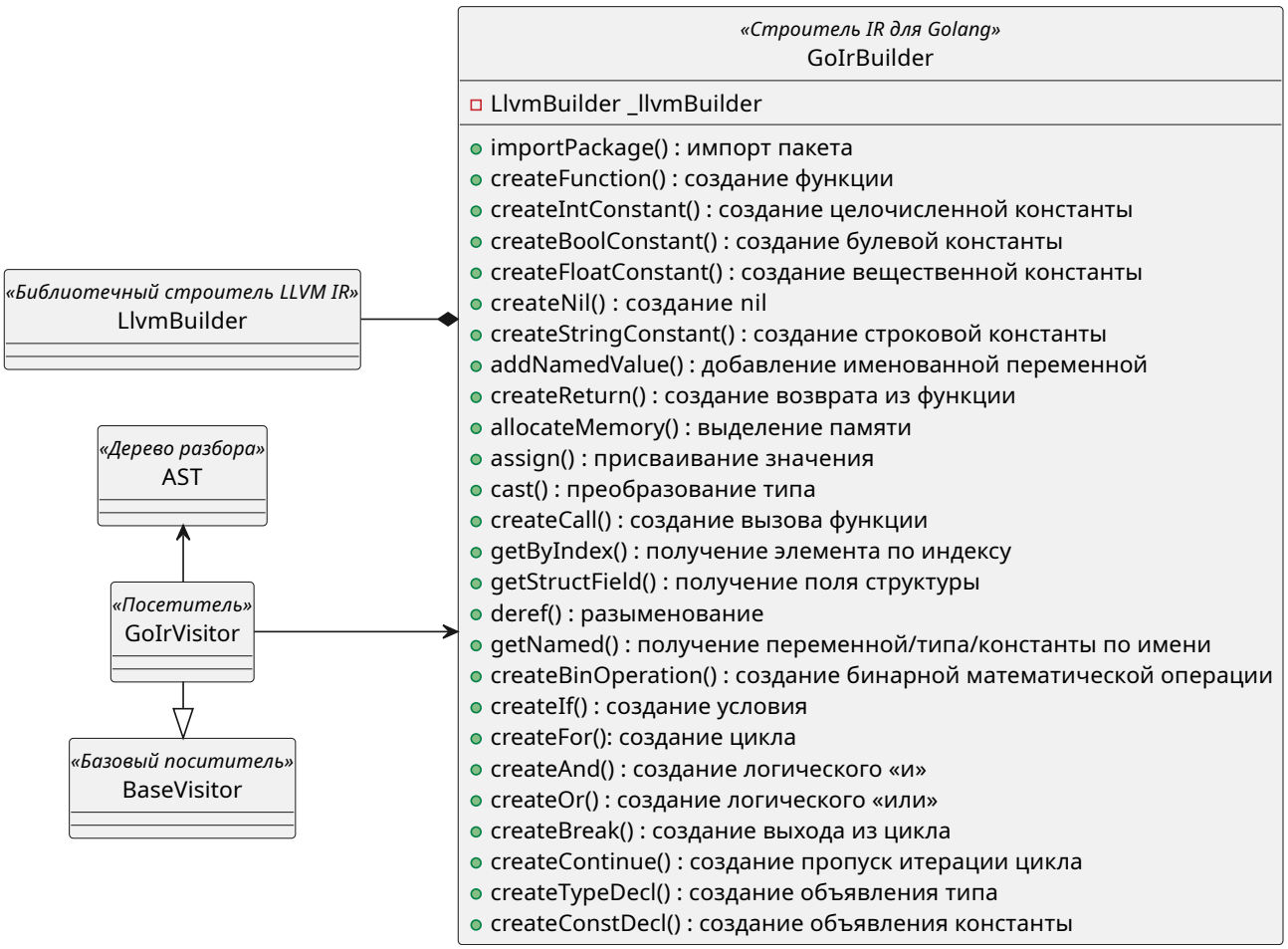

Рисунок 2.3 — Структура строителя IR представления

Для каждого выделенного объекта предлагается хранить его разметку, где указывается местоположения указателей объекта. Таким образом можно получить ссылки объекта, зная только информацию о его разметке в памяти. Также для дальнейшей очистки неиспользуемых объектов предлагается использовать флаг об использовании.

На рисунке 2.4 показано представление в памяти теневых стека и кучи.

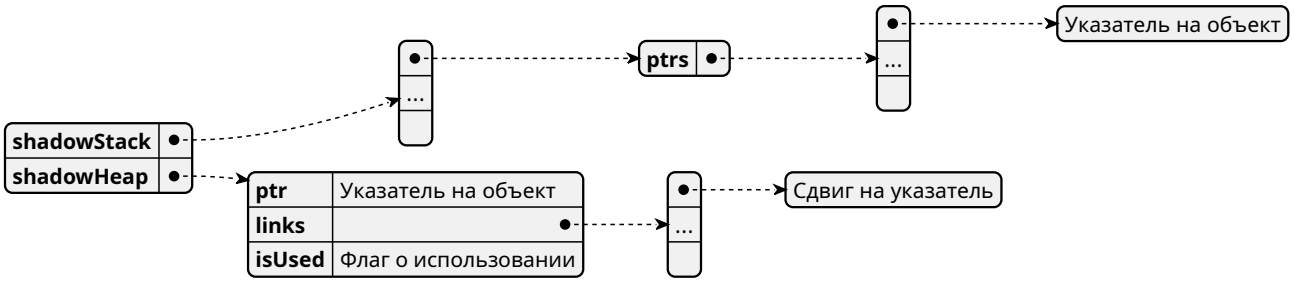

Рисунок 2.4 — Представление в памяти теневых стека и кучи

Алгоритм 2.2 предназначен для выделения памяти под новый объект в куче. На вход функция алгоритма принимает размер выделяемого объекта в байтах и массив ссылок на другие объекты, указывающих на начало указателей в объекте. Внутри функции сначала выделяется блок памяти заданного размера и очищается. Затем адрес выделенной памяти добавляется в конец

глобального теневого стека, а в хэш-таблицу кучи по адресу объекта добавляется информация о его разметке. В конце функция возвращает указатель на выделенную память.

Алгоритм 2.2 — Создание нового объекта

1: **Функция** ALLOCATE(size, links)
   ▷ $size$ — размер выделяемого типа в байтах
   ▷ $links$ — массив позиций в типе, указывающих на начало ссылок на иные объекты
   ▷ $shadowStack$ — глобальный теневой стек
   ▷ $shadowHeap$ — глобальная хэш-таблица: адрес памяти / информация о ссылках
2: $ptr \leftarrow$ выделить $size$ байт и очистить их
3: добавить адрес ptr в последний блок теневого стека $shadowStack$
4: добавить по ключу ptr значение $links$ в таблицу $shadowHeap$
5: **Вернуть** ptr
6: **Конец функции**

Сбор неиспользуемой памяти

Алгоритм 2.3 предназначен для освобождения памяти, которая больше не используется программой. На вход алгоритму подается массив указателей ptrs, содержащий адреса дополнительно сохраняемых объектов. В данном случае – это значения, возвращаемые последней функцией, так как они могли еще не сохраниться в какой-либо переменной на стеке и могут быть очищены. Происходит проход по каждому сохраняемого объекту и вызывается функция Mark, которая помечает объекты, как используемые. Затем выполняется проход по глобальному теневому стеку shadowStack, чтобы пометить объекты стека как используемые. После завершения данного процесса все объекты, на которые не указывают ни массив ptrs, ни глобальный теневой стек, освобождаются. Освобождение памяти происходит путем удаления объекта из глобальной хэш-таблицы shadowHeap и отправка запроса на удаление алокатору.

Алгоритм 2.3 — Сбор неиспользуемой памяти

1: **Функция** $GarbageCollection(ptr)$
   ▷ $ptrs$ — массив указателей на сохраняемые объекты
   ▷ $shadowHeap$ — глобальная хэш-таблица: адрес памяти / информация о ссылках
   ▷ $shadowStack$ — глобальный теневой стек
2: **Для всех** $ptr \in ptrs$ **выполнить**
3: $Mark(ptr)$
4: **Конец цикла**
5: **Для всех** $layer \in shadowStack$ **выполнить**

6: **Для всех** $ptr \in layer$ **выполнить**
7: $Mark(ptr)$
8: **Конец цикла**
9: **Конец цикла**
10: **Для всех** $ptr \in shadowHeap$, которые не помечены как используемые **выполнить**
11: Освободить память ptr
12: **Конец цикла**
13: **Конец функции**

Алгоритм 2.4 предназначен для пометки объекта и всех объектов, на которые он ссылается, как используемые. На вход алгоритму передается указатель на объект, который необходимо пометить. Сначала проверяется, не является ли указатель пустым. Затем из глобальной хэш-таблицы кучи получается информация о ссылках объекта и для каждой ссылки вызывается рекурсивно данная процедура.

Алгоритм 2.4 — Пометить объект и всех объектов, на которые он ссылается

1: **Функция** $Mark(ptr)$
▷ ptr — указатель на раскрашиваемый объект
▷ $shadowHeap$ — глобальная хэш-таблица: адрес памяти / информация о ссылках

2: **Если** ptr – пустой **тогда**
3: **Вернуть**
4: **Конец условия**
5: $links \leftarrow$ получить значение из $shadowHeap$ по ключу ptr
6: **Если** не удалось получить значение $links$ **тогда**
7: вывести сообщение о наличии «дикого» указателя
8: **Вернуть**
9: **Конец условия**
10: **Если** ptr помечен как используемый **тогда**
11: **Вернуть**
12: **Конец условия**
13: пометить ptr как используемый
14: **Для всех** $link \in links$ **выполнить**
15: $linkPtr \leftarrow$ получить значение по адресу $ptr + link$
16: $Mark(linkPtr)$
17: **Конец цикла**
18: **Вернуть**
19: **Конец функции**

С помощью предложенных алгоритмов реализуются основные процессы выделения и утилизации памяти.

Выводы

Была разработана концептуальная модель компилятора языка программирования Golang. Предложены способы представления пакетов и их предварительной обработки. Спроектирована система представления типов, покрывающая грамматику языка. Рассмотрены вопросы построения абстрактного синтаксического дерева программы. Структура строителя промежуточного представления кода была спроектирована. Были описаны алгоритмы управления памяти и предложены сущности для хранения информации об используемых объектах и их структуры.

3 Технологический раздел

3.1 Выбор средств реализации

В ходе разработки было решено использовать язык программирования C++, так как он предоставляет широкие возможности для реализации приложений. Язык программирования C++ широко поддерживается компилятором LLVM, что обеспечивает высокую производительность и эффективность разработки. Кроме того, ANTLR4, используемый для генерации кода лексера и парсера, также порождает исходный код на C++, что позволяет легко интегрировать сгенерированный код. Также код на C++ легко компонуется с компилируемыми бекендом LLVM программами, благодаря поддержке соглашения вызовов cdecl, что облегчает процесс разработки и интеграции. Для обеспечения удобного импорта и экспорта данных о структуре пакетов была выбрана библиотека Boost, обладающая нужными инструментами для работы с данными. Для написания кода были использованы интегрированные среды разработки JetBrains CLion и GoLand.

Структура проекта

На рисунке 3.1 представлен граф включаемых заголовочных файлов для main.cpp.

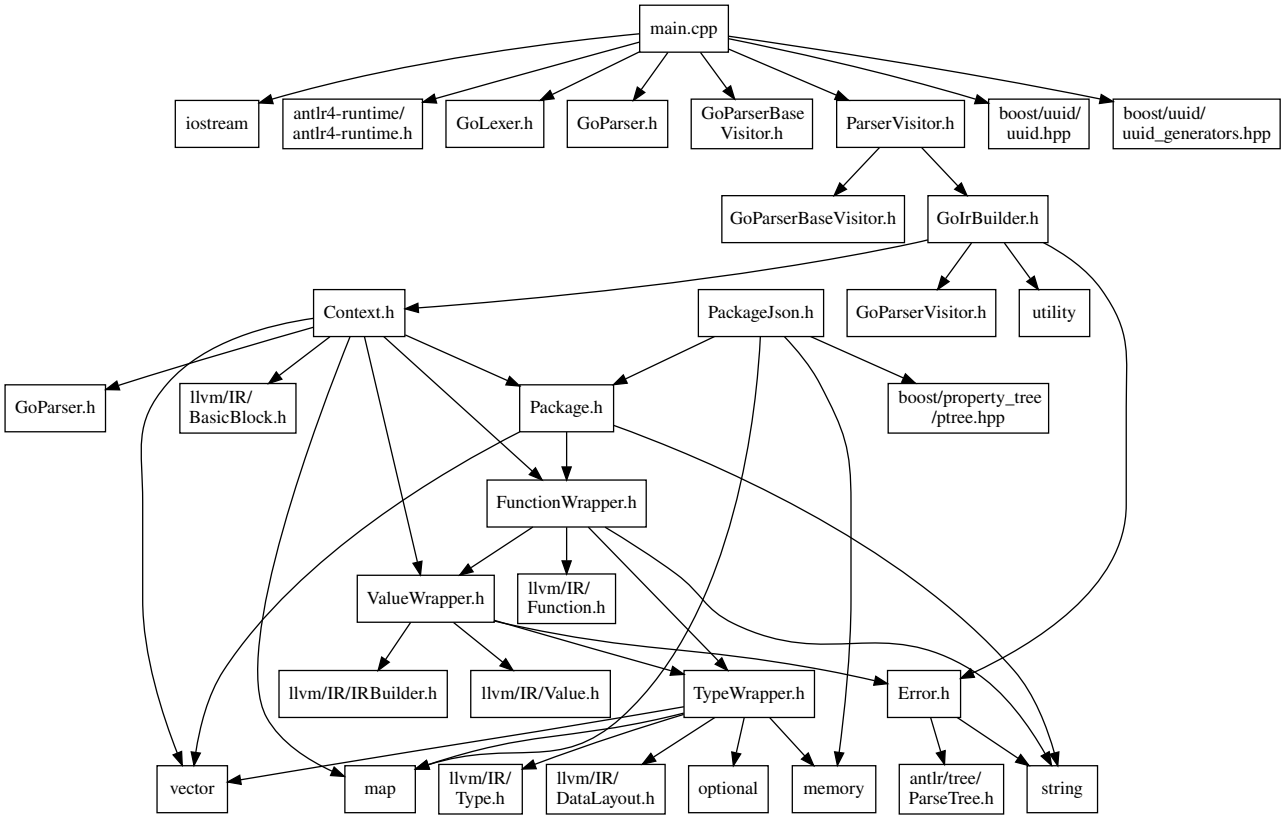

Рисунок 3.1 — Граф включаемых заголовочных файлов для main.cpp

3.2 Тестирования компилятора

Тестирование генерации кода

Были созданы программы, которые соответствуют измененной грамматике языка. Они предназначены для проверки правильности работы различных конструкций языка. Данные программы охватывают следующие сценарии:

— создание переменных и констант;

— ввод и вывод;

— работа с целыми и вещественными числами;

— работа с константными строками;

— создание функций и работа с рекурсией;

— создание ветвлений программы;

— создание циклов, выход из цикла и пропуск итерации цикла;

— работа с массивами фиксированного размера;

— работа со структурами, создание методов структур;

— создание интерфейсов;

— работа с указателями;

— логические операции, ленивые логические вычисления;

— преобразование типов.

В листинге 3.1 представлен пример работы с интерфейсами.

Листинг 3.1 — Пример программы использования интерфейсов

```
1  package main
2  import "fmt"
3  type geometry interface {
4      area() float64
5      perim() float64
6  }
7  type rect struct {
8      width  float64
9      height float64
10 }
11 type circle struct {
12     radius float64
13 }
14 func (r rect) area() float64 {
15     return r.width * r.height
16 }
17 func (r rect) perim() float64 {
18     return 2*r.width + 2*r.height
```

```
}
func (c circle) area() float64 {
    return 3.14 * c.radius * c.radius
}
func (c circle) perim() float64 {
    return 2 * 3.14 * c.radius
}
func measure(g geometry) {
    fmt.Printf("Area: %g cm^2, perim: %g cm\n", g.area(), g.perim())
}
func main() {
    var r rect
    r.width = float64(3)
    r.height = float64(4)
    var c circle
    c.radius = float64(5)
    var g1 geometry = r
    measure(g1)
    var g2 geometry = c
    measure(g2)
}
```

Было обнаружено различие в работе данного и оригинального компилятора, заключающеюся в форматировании булевых значений. Оно связано с тем, что используются разные реализации функции printf. Данное различие несущественно и может быть опущено из рассмотрения. Компилятор справился со всеми описанными задачами успешно.

Тестирование работы с памятью

Для тестирования сборщика мусора был использован инструмент Valgrind с использованием предыдущих программ.

Рассмотрим следующий пример, представленный на листинге 3.2. В программе создается связанный список из 1000 узлов. Каждый узел занимает примерно 390 КиБ. Затем голова списка обнуляется, чтобы ссылки на все узлы стали недоступными для дальнейшего использования. Затем создается связанный список из 200 узлов. Данные операции были повторены 10 раз. Результат работы сборщика мусора можно оценить, наблюдая изменения в использовании памяти программой.

Листинг 3.2 — Программа для тестирования сборщика мусора

```
package main
type Node struct {
    data     [100000]int
    nextNode *Node
}
func newNode(next *Node) *Node {
    var node Node
    node.nextNode = next
    return &node
}
func main() {
```

```
12      var head *Node = nil
13      for i := 0; i < 10; i++ {
14          for i := 0; i < 1000; i++ {
15              head = newNode(head)
16          }
17          head = nil
18          for i := 0; i < 200; i++ {
19              head = newNode(head)
20          }
21          head = nil
22      }
23  }
```

Инструмент Valgrind не выявил ошибок в использовании памяти. Также был получен график использования памяти, представленный на рисунке 3.2. Из данной зависимости видно, что сборщик мусора высвобождал неиспользуемые узлы. Следовательно, сборщик мусора работает корректно.

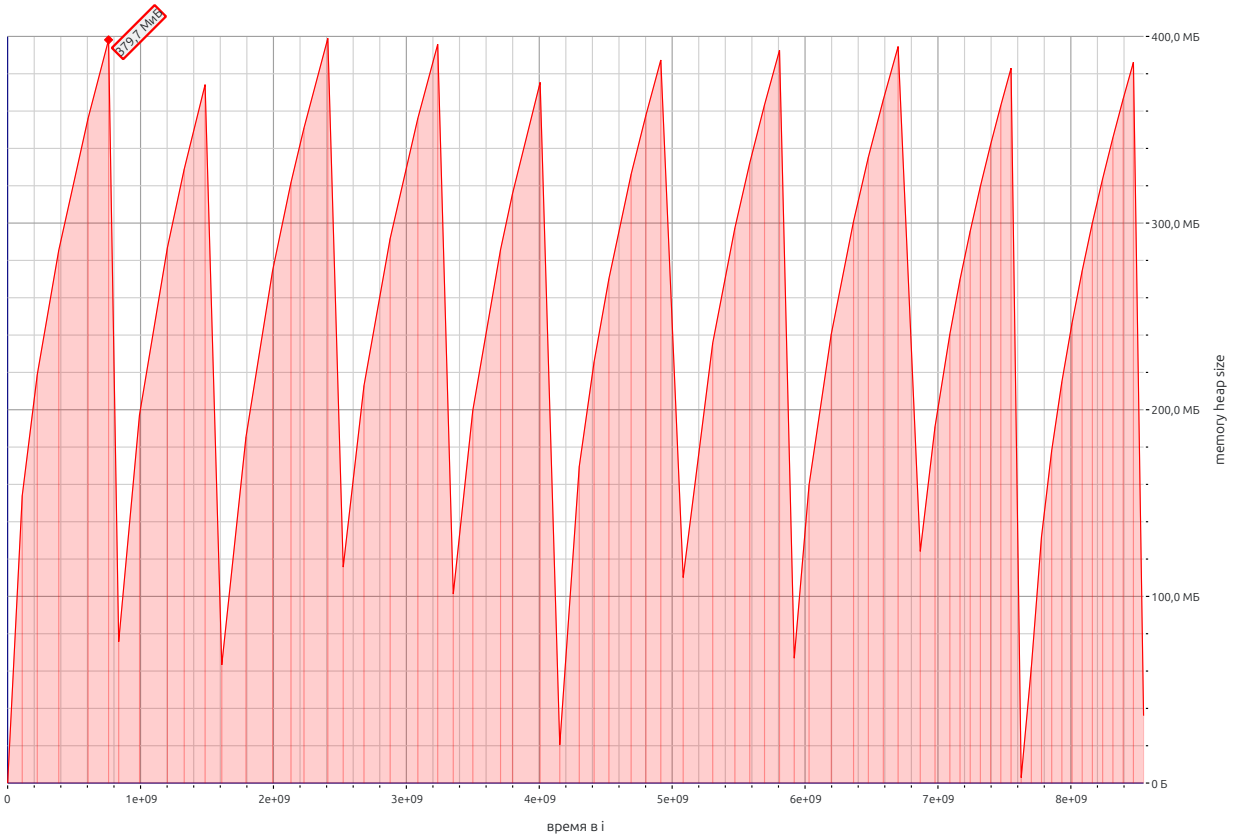

Рисунок 3.2 — Зависимость используемой памяти от времени для программы, представленной на листинге 3.2

ЗАКЛЮЧЕНИЕ

В результате проведенной работы был осуществлен анализ предметной области разработки компилятора языка программирования Golang. Были рассмотрены ключевые аспекты, включая анализ инфраструктуры, сравнительное изучение инструментов для генерации лексических и синтаксических анализаторов, а также аспекты типизации, управления ресурсами и организации пакетов в данном языке. Данный анализ позволил сформулировать формальную задачу разработки компилятора Golang.

В ходе исследования была разработана концептуальная модель компилятора, определены методы представления пакетов и их предварительной обработки, а также спроектирована система представления типов, охватывающая грамматику языка. Кроме того, были изучены аспекты построения абстрактного синтаксического дерева программы, спроектирована структура строителя промежуточного представления кода, и разработаны алгоритмы управления памятью, предложены сущности для хранения информации об используемых объектах и их структуры.

Выбор языка программирования C++ для реализации проекта обусловлен его мощными возможностями и поддержкой компилятором LLVM. Разработанные программы для тестирования функциональности компилятора охватывают разнообразные языковые конструкции, а тестирование сборщика мусора с использованием инструментов анализа использования памяти позволяет удостовериться в его работоспособности.

Поставленные задачи были решены. Цель работы достигнута: был разработан прототип компилятора Golang. Полученные результаты являются основой для дальнейшей работы над развитием и оптимизацией данного компилятора.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Компиляторы: принципы, технологии и инструментарий / Ахо А, Лам М, Сети Рави и Ульман Дж // М.: Вильямс. — 2008. — Т. 1. — С. 257.

2. Bodner Jon. Learning Go. — "O'Reilly Media, Inc. 2024.

3. Wilhelm Reinhard, Maurer Dieter и др. Compiler design. — Springer, 1995.

4. Ахо А Ульман Дж. Теория синтаксического анализа, перевода и компиляции. Синтаксический анализ. — 1978.

5. Кадомский Андрей Андреевич, Сабинин Олег Юрьевич. ИССЛЕДОВАНИЕ ВОЗМОЖНОСТИ СОЗДАНИЯ УНИВЕРСАЛЬНОГО ЯЗЫКА ПРОГРАММИРОВАНИЯ ВЫСОКОГО УРОВНЯ // Theoretical & Applied Science. — 2018. — № 5. — С. 84–90.

6. Sippu Seppo, Soisalon-Soininen Eljas. Parsing Theory: Volume II LR (k) and LL (k) Parsing. — Springer Science & Business Media, 2013. — Т. 20.

7. Lesk Michael E, Schmidt Eric. Lex: A lexical analyzer generator. — Bell Laboratories Murray Hill, NJ, 1975. — Vol. 39.

8. Paxson Vern. Flex – A fast scanner generator Edition. — 1995. — Режим доступа: `https://ftp.gnu.org/old-gnu/Manuals/flex-2.5.4/html_mono/flex.html` (дата обращения: 20.03.2024).

9. Parr Terence. The definitive ANTLR 4 reference // The Definitive ANTLR 4 Reference. — 2013. — С. 1–326.

10. Hanspeter Mössenböck, Löberbauer Markus, Albrecht Wö■. The Compiler Generator Coco/R. — 2018. — Режим доступа: `https://ssw.jku.at/Research/Projects/Coco/` (дата обращения: 20.03.2024).

11. Appel Andrew Wilson. A Runtime System // Journal of Lisp and Symbolic Computation 3. — 1990. — С. 343–380.

12. Foundation LLVM. The LLVM Compiler Infrastructure Project. — 2024. — Режим доступа: `https://llvm.org/` (дата обращения: 21.03.2024).

13. Vichare Abhijat. The Conceptual Structure of GCC // Indian Institute of Technology, Bombay. — 2008.

14. Bruno Rodrigo, Ferreira Paulo. A study on garbage collection algorithms for big data environments // ACM Computing Surveys (CSUR). — 2018. — Т. 51, № 1. — С. 1–35.

15. Appel Andrew W. Simple generational garbage collection and fast allocation // Software: Practice and experience. — 1989. — Т. 19, № 2. — С. 171–183.

16. A study of the scalability of stop-the-world garbage collectors on multicores / Gidra Lokesh, Thomas Gaël, Sopena Julien и Shapiro Marc // ACM SIGPLAN Notices. — 2013. — Т. 48, № 4. — С. 229–240.

17. Collins George E. A method for overlapping and erasure of lists // Communications of the ACM. — 1960. — Т. 3, № 12. — С. 655–657.

18. McCarthy John. Recursive functions of symbolic expressions and their computation by machine, part I // Communications of the ACM. — 1960. — Vol. 3, no. 4. — P. 184–195.

ПРИЛОЖЕНИЕ А

ПРЕДСТАВЛЕНИЕ СТАНДАРТНОЙ БИБЛИОТЕКИ

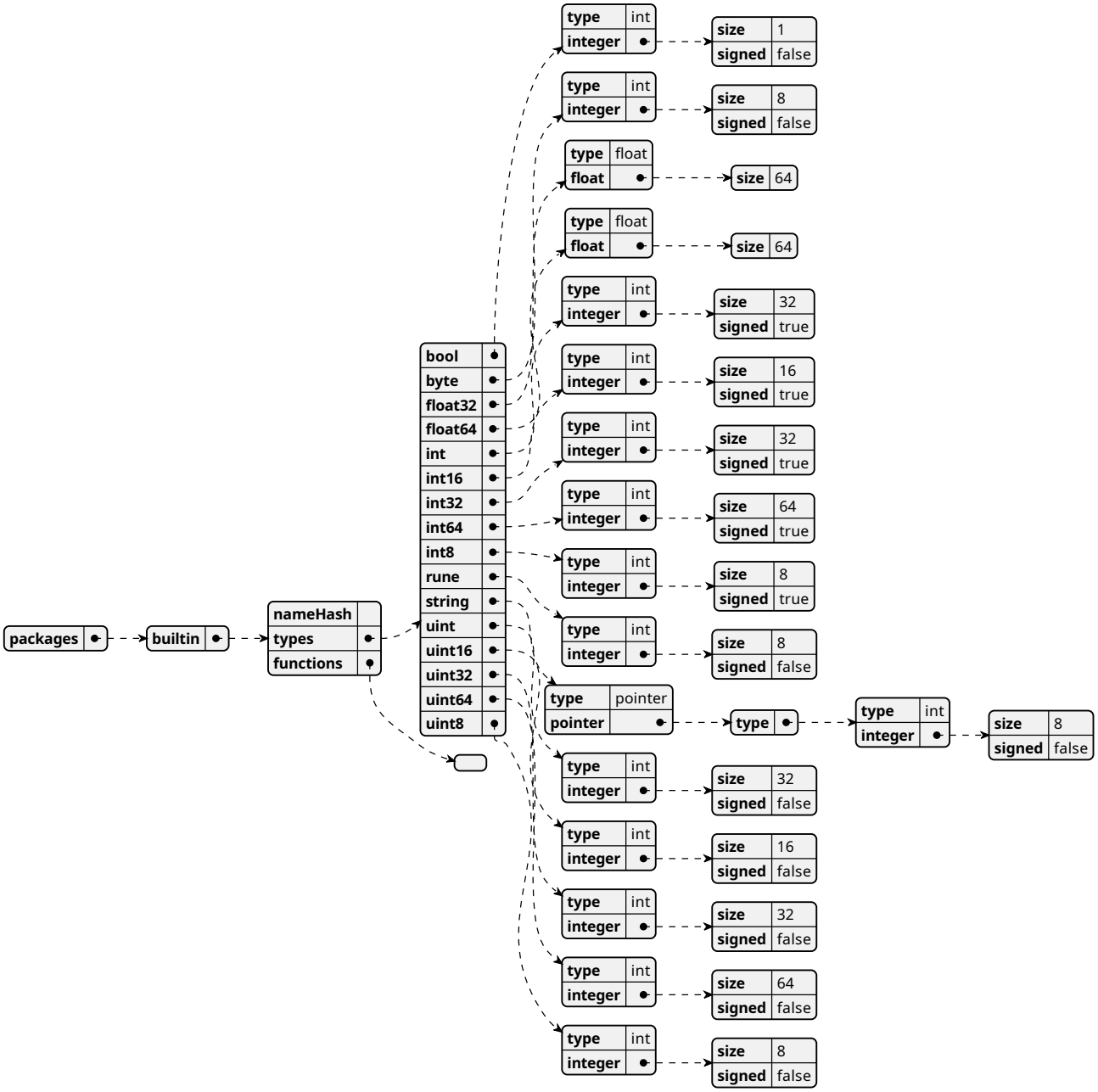

Рисунок А.1 — Представление стандартной библиотеки

ПРИЛОЖЕНИЕ Б

ГРАММАТИКА ЯЗЫКА ПРОГРАММИРОВАНИЯ GOLANG

Листинг Б.1 — Описание лексического анализатора

```
lexer grammar GoLexer;
BREAK : 'break' -> mode(NLSEMI);
FUNC : 'func';
INTERFACE : 'interface';
STRUCT : 'struct';
ELSE : 'else';
PACKAGE : 'package';
CONST : 'const';
IF : 'if';
RANGE : 'range';
TYPE : 'type';
CONTINUE : 'continue' -> mode(NLSEMI);
FOR : 'for';
IMPORT : 'import';
RETURN : 'return' -> mode(NLSEMI);
VAR : 'var';
NIL_LIT: 'nil' -> mode(NLSEMI);
IDENTIFIER: LETTER (LETTER | UNICODE_DIGIT)* -> mode(NLSEMI);
L_PAREN : '(';
R_PAREN : ')' -> mode(NLSEMI);
L_CURLY : '{';
R_CURLY : '}' -> mode(NLSEMI);
L_BRACKET : '[';
R_BRACKET : ']' -> mode(NLSEMI);
ASSIGN : '=';
COMMA : ',';
SEMI : ';';
COLON : ':';
DOT : '.';
PLUS_PLUS : '++' -> mode(NLSEMI);
MINUS_MINUS : '--' -> mode(NLSEMI);
DECLARE_ASSIGN : ':=';
ELLIPSIS : '...';
LOGICAL_OR : '||';
LOGICAL_AND : '&&';
EQUALS : '==';
NOT_EQUALS : '!=';
LESS : '<';
LESS_OR_EQUALS : '<=';
GREATER : '>';
GREATER_OR_EQUALS : '>=';
EXCLAMATION: '!';
PLUS : '+';
MINUS : '-';
STAR : '*';
AMPERSAND : '&';
DECIMAL_LIT : ('0' | [1-9] ('_'? [0-9])*) -> mode(NLSEMI);
FLOAT_LIT: (DECIMAL_FLOAT_LIT ) -> mode(NLSEMI);
DECIMAL_FLOAT_LIT: DECIMALS ('.' DECIMALS? EXPONENT? | EXPONENT) | '.' DECIMALS EXPONENT?;
fragment RUNE: '\'' (UNICODE_VALUE) '\'';
RUNE_LIT: RUNE -> mode(NLSEMI);
INTERPRETED_STRING_LIT : '"' (~["\\] | ESCAPED_VALUE)* '"' -> mode(NLSEMI);
WS : [ \t]+ -> channel(HIDDEN);
```

```
54 COMMENT : '/*' .*? '*/' -> channel(HIDDEN);
55 TERMINATOR : [\r\n]+ -> channel(HIDDEN);
56 LINE_COMMENT : '//' ~[\r\n]* -> channel(HIDDEN);
57 fragment UNICODE_VALUE: ~[\r\n'] | LITTLE_U_VALUE | BIG_U_VALUE ;
58 fragment DECIMALS: [0-9] ('_'? [0-9])*;
59 fragment EXPONENT: [eE] [+-]? DECIMALS;
60 fragment LETTER: UNICODE_LETTER | '_';
61 fragment UNICODE_DIGIT: [\p{Nd}];
62 fragment UNICODE_LETTER: [\p{L}];
63 mode NLSEMI;
64 WS_NLSEMI: [ \t]+ -> channel(HIDDEN);
65 COMMENT_NLSEMI : '/*' ~[\r\n]*? '*/' -> channel(HIDDEN);
66 LINE_COMMENT_NLSEMI : '//' ~[\r\n]* -> channel(HIDDEN);
67 EOS: ([\r\n]+ | ';' | '/*' .*? '*/' | EOF) -> mode(DEFAULT_MODE);
68 OTHER: -> mode(DEFAULT_MODE), channel(HIDDEN);
```

Листинг Б.2 — Описание синтаксического анализатора

```
1  parser grammar GoParser;
2  options {
3  tokenVocab = GoLexer;
4  superClass = GoParserBase;
5  }
6  sourceFile : packageClause eos (importDecl eos)* ((functionDecl | methodDecl | declaration)
       eos)* EOF ;
7  packageClause : PACKAGE packageName = IDENTIFIER ;
8  importDecl : IMPORT (importSpec | L_PAREN (importSpec eos)* R_PAREN) ;
9  importSpec : alias = (DOT | IDENTIFIER)? importPath ;
10 importPath : string_ ;
11 declaration : constDecl | typeDecl | varDecl ;
12 constDecl : CONST (constSpec | L_PAREN (constSpec eos)* R_PAREN) ;
13 constSpec : identifierList (type_? ASSIGN expressionList)? ;
14 identifierList : IDENTIFIER (COMMA IDENTIFIER)* ;
15 expressionList : expression (COMMA expression)* ;
16 typeDecl : TYPE (typeSpec | L_PAREN (typeSpec eos)* R_PAREN) ;
17 typeSpec : aliasDecl | typeDef ;
18 aliasDecl : IDENTIFIER ASSIGN type_ ;
19 typeDef : IDENTIFIER typeParameters? type_ ;
20 typeParameters : L_BRACKET typeParameterDecl (COMMA typeParameterDecl)* R_BRACKET ;
21 typeParameterDecl : identifierList typeElement ;
22 typeElement : typeTerm (OR typeTerm)* ;
23 typeTerm : UNDERLYING? type_ ;
24 functionDecl : FUNC IDENTIFIER typeParameters? signature block? ;
25 methodDecl : FUNC receiver IDENTIFIER signature block? ;
26 receiver : parameters ;
27 varDecl : VAR (varSpec | L_PAREN (varSpec eos)* R_PAREN) ;
28 varSpec : identifierList (type_ (ASSIGN expressionList)? | ASSIGN expressionList) ;
29 block : L_CURLY statementList? R_CURLY ;
30 statementList : ((SEMI? | EOS? | {this->closingBracket()}?) statement eos)+ ;
31 statement : declaration | simpleStmt | returnStmt | breakStmt | continueStmt | block | ifStmt |
       forStmt ;
32 simpleStmt : incDecStmt | assignment | expressionStmt | shortVarDecl ;
33 expressionStmt : expression ;
34 incDecStmt : expression (PLUS_PLUS | MINUS_MINUS) ;
35 assignment : expressionList assign_op expressionList ;
36 assign_op : (PLUS | MINUS | OR | CARET | STAR | DIV | MOD | AMPERSAND )? ASSIGN ;
37 shortVarDecl : identifierList DECLARE_ASSIGN expressionList ;
38 returnStmt : RETURN expressionList? ;
39 breakStmt : BREAK IDENTIFIER? ;
```

```
40 continueStmt : CONTINUE IDENTIFIER? ;
41 ifStmt : IF (expression | eos expression | simpleStmt eos expression) block (ELSE (ifStmt |
       block))? ;
42 typeList : (type_ | NIL_LIT) (COMMA (type_ | NIL_LIT))* ;
43 forStmt : FOR (expression? | forClause | rangeClause?) block ;
44 forClause : initStmt = simpleStmt? eos expression? eos postStmt = simpleStmt? ;
45 rangeClause : (expressionList ASSIGN | identifierList DECLARE_ASSIGN)? RANGE expression ;
46 type_ : typeName typeArgs? | typeLit | L_PAREN type_ R_PAREN ;
47 typeArgs : L_BRACKET typeList COMMA? R_BRACKET ;
48 typeName : qualifiedIdent | IDENTIFIER ;
49 typeLit : arrayType | structType | pointerType | functionType | interfaceType ;
50 arrayType : L_BRACKET arrayLength R_BRACKET elementType ;
51 arrayLength : expression ;
52 elementType : type_ ;
53 pointerType : STAR type_ ;
54 interfaceType : INTERFACE L_CURLY ((methodSpec | typeElement) eos)* R_CURLY ;
55 methodSpec : IDENTIFIER parameters result | IDENTIFIER parameters ;
56 functionType : FUNC signature ;
57 signature : parameters result? ;
58 result : parameters | type_ ;
59 parameters : L_PAREN (parameterDecl (COMMA parameterDecl)* COMMA?)? R_PAREN ;
60 parameterDecl : identifierList? ELLIPSIS? type_ ;
61 expression : primaryExpr | unary_op = (PLUS | MINUS | EXCLAMATION | CARET | STAR | AMPERSAND |
       RECEIVE) expression | expression mul_op = (STAR | DIV | MOD | LSHIFT | RSHIFT | AMPERSAND
       | BIT_CLEAR) expression | expression add_op = (PLUS | MINUS | OR | CARET) expression |
       expression rel_op = (
62 EQUALS | NOT_EQUALS | LESS | LESS_OR_EQUALS | GREATER | GREATER_OR_EQUALS
63 ) expression | expression LOGICAL_AND expression | expression LOGICAL_OR expression ;
64 primaryExpr : operand | conversion | methodExpr | primaryExpr ( DOT IDENTIFIER | index | slice_
       | typeAssertion | arguments) ;
65 conversion : type_ L_PAREN expression COMMA? R_PAREN ;
66 operand : literal | operandName // typeArgs? | L_PAREN expression R_PAREN ;
67 literal : basicLit | functionLit ;
68 basicLit : NIL_LIT | integer | string_ | FLOAT_LIT ;
69 integer : DECIMAL_LIT | RUNE_LIT ;
70 operandName : IDENTIFIER ;
71 qualifiedIdent : IDENTIFIER DOT IDENTIFIER ;
72 structType : STRUCT L_CURLY (fieldDecl eos)* R_CURLY ;
73 fieldDecl : (identifierList type_ | embeddedField) tag = string_? ;
74 string_ : RAW_STRING_LIT | INTERPRETED_STRING_LIT ;
75 embeddedField : STAR? typeName typeArgs? ;
76 functionLit : FUNC signature block ; // function
77 index : L_BRACKET expression R_BRACKET ;
78 slice_ : L_BRACKET (expression? COLON expression? | expression? COLON expression COLON
       expression) R_BRACKET ;
79 typeAssertion : DOT L_PAREN type_ R_PAREN ;
80 arguments : L_PAREN ((expressionList | type_ (COMMA expressionList)?) ELLIPSIS? COMMA?)?
       R_PAREN ;
81 methodExpr : type_ DOT IDENTIFIER ;
```