1. Министерство науки и высшего образования Российской Федерации
2. Санкт-Петербургский Политехнический Университет Петра Великого
3. —

Институт компьютерных наук и кибербезопасности

**КУРСОВАЯ РАБОТА**

1. по дисциплине «Формальные грамматики и теория компиляторов»
2. Выполнили: студенты группы
   1. 5151003/10801 Суранов Н. И.

<*подпись*>

* 1. Якушев Н. Д.

<*подпись*>

1. Проверил преподаватель Грибков Н. А.

<*подпись*>

1. Санкт-Петербург
2. 2024

Оглавление

[ЦЕЛИ 3](#_Toc165915762)

[ЗАДАЧИ 3](#_Toc165915763)

[ВВЕДЕНИЕ 4](#_Toc165915764)

[Анализатор языка 4](#_Toc165915765)

[История создания Golang 4](#_Toc165915766)

[ХОД РАБОТЫ 7](#_Toc165915767)

[Синтаксис Golang 7](#_Toc165915768)

[Основные конструкции языка 7](#_Toc165915769)

[Тестирование 17](#_Toc165915770)

[ВЫВОД 20](#_Toc165915771)

[ПРИЛОЖЕНИЕ 20](#_Toc165915772)

[Flex 20](#_Toc165915773)

[Bison 24](#_Toc165915774)

1. **ЦЕЛИ**

Создание грамматики существующего формального языка (Golang).

# **ЗАДАЧИ**

Необходимо разработать распознаватель любого современного формального языка на ваш выбор. Можно брать:

1. Любые языки программирования (кроме Паскаля).
2. Специализированные языки Unix-утилит, в том числе awk, make, yacc и т.п.
3. Языки, применяемые в компьютерной безопасности, например язык описания атак NASL из Nessus.
4. Языки командных интерпретаторов, типа bash или PowerShell.
5. Типографские языки, например, Latex, Ppostscript.
6. Языки разметки: HTML, XML.

Разумно брать язык, который вы хорошо знаете/хотите изучить. Выбранные языки в пределах группы не должны повторяться.

Объем выполняемой работы зависит от сложности. Для сложных языков достаточно сделать распознаватель с минимальными сообщениями об ошибках, для простых языков нужно будет что-то еще, например, верификацию, генерацию кода или форматирование.

Работы выполняются в парах. Допускается объединение в команды по 3 человека, если объем работы большой. Если возникают трудности с определением объема работы для выбранного языка и количества человек, можно задать их на почту либо подойти очно с предварительным уведомлением по почте.

Основное требование - самостоятельно написать грамматику, пусть несовершенную, но свою.

Грамматики, взятые из интернета, приниматься не будут.

# **ВВЕДЕНИЕ**

## **Анализатор языка**

Грамматический разбор представляет собой процедуру сопоставления последовательности слов или лексем языка с его формальной грамматикой. В результате обычно формируется дерево разбора или абстрактное синтаксическое дерево. В этом процессе применяются грамматические правила для определения и распознавания последовательности лексем, а также для выполнения соответствующих операций. Для создания грамматического анализатора используется инструмент bison, который интерпретирует описание грамматики и создает распознаватель для конкретного языка.

Лексический анализатор является компонентом компилятора, который обрабатывает исходный код программы и выделяет из него лексемы. Он принимает на вход текст программы и результатом его работы является список всех обнаруженных лексем. Для реализации лексического анализатора применяется инструмент Flex. На начальном этапе компиляции Flex читает входные данные и трансформирует строки исходного кода в токены, где каждому шаблону соответствует определенное действие. Обычно, в результате действия возвращается токен, который представляет распознанную строку и используется далее в анализаторе.

## **История создания Golang**

Язык программирования Go, также известный как Golang, был создан в компании Google в 2007 году командой разработчиков, в которую входили Роберт Гриземер, Роб Пайк и Кен Томпсон. Основная мотивация для создания нового языка заключалась в желании устранить недостатки, которые они видели в других языках программирования при использовании в масштабах больших систем и многопоточных программ, что часто встречалось в работе в Google.

Язык программирования Go, также известный как Golang, был создан в компании Google в 2007 году командой разработчиков, в которую входили Роберт Гриземер, Роб Пайк и Кен Томпсон. Основная мотивация для создания нового языка заключалась в желании устранить недостатки, которые они видели в других языках программирования при использовании в масштабах больших систем и многопоточных программ, что часто встречалось в работе в Google.

Цели создания Go

1. **Простота и эффективность**: Создатели стремились разработать язык, который был бы прост в изучении и использовании, но в то же время эффективным для решения практических задач разработки программного обеспечения.
2. **Быстрая компиляция**: Одной из ключевых целей было обеспечение высокой скорости компиляции, чтобы ускорить процесс разработки.
3. **Поддержка конкурентности**: Язык должен был нативно поддерживать конкурентные операции, делая написание многопоточных программ более доступным и безопасным.
4. **Работа с многопроцессорными системами**: Важной целью была поддержка эффективного параллельного выполнения программ на современных многопроцессорных и распределенных системах.

Go активно используется во многих областях разработки программного обеспечения, благодаря своей производительности, простоте и поддержке конкурентного программирования. Вот основные области его применения:

1. **Веб-разработка**: Go широко применяется для создания веб-серверов и веб-сервисов благодаря своей способности обрабатывать большое количество соединений и запросов одновременно. Стандартная библиотека Go включает в себя мощные инструменты для работы с HTTP, что делает его подходящим для разработки RESTful API и микросервисов.
2. **Микросервисы**: Язык идеально подходит для создания микросервисных архитектур, поскольку он обладает нативной поддержкой конкурентности и легкостью развертывания. Компании, такие как Netflix и Uber, используют Go для построения своих микросервисных платформ. Нередки случаи, когда отдельные, высоконагруженные части какого-то серверного приложения переписывают на golang, чтобы ускорить его работу.
3. **Сетевое программирование**: благодаря простоте создания многопоточных программ с использованием горутин и каналов, Go стал популярен для разработки сетевых служб, таких как серверы, прокси и другие инструменты, требующие высокой производительности и одновременной обработки множества соединений.
4. **Облачные сервисы и инфраструктура**: Go особенно популярен в облачных платформах и инфраструктуре. Google Cloud, AWS и другие крупные провайдеры используют Go для создания инструментов и сервисов. Kubernetes, ведущая система оркестрации контейнеров, написана на Go.
5. **DevOps и автоматизация**: Инструменты для автоматизации и управления инфраструктурой, такие как Terraform и Docker, часто используют Go из-за его эффективности и способности легко распространяться как один исполняемый файл.
6. **Командные утилиты и скрипты**: Go также применяется для создания исполняемых командных утилит и скриптов благодаря быстрому времени выполнения и возможности компиляции в один статический файл, что упрощает распространение и использование.
7. **Игры и развлекательные приложения**: Некоторые разработчики используют Go для создания серверной части игр, где требуется высокая производительность и обработка параллельных задач.

# **ХОД РАБОТЫ**

## **Синтаксис Golang**

Синтаксис Go во многом напоминает C, но с упрощениями и дополнениями, которые делают его более безопасным и удобным для использования в современной разработке программного обеспечения. Рассмотрим основные особенности синтаксиса и нюансы языка Go.

## **Основные конструкции языка**

Разберем основные конструкции в golang и как мы их реализовали.

В первую очередь посмотрим базовую структуру любой программы на golang. В первую очередь у нас идет название пакета, затем блок импортов, и после них идут те сущности, которые мы назвали default\_block – это могут быть объявления структур, интерфейсов, методов, функций или присваивание констант или переменных.

Название пакета представляет из себя всего два слова: package name. Где name – название пакета (например, main) и package – ключевое слово, которое установлено у нас как токен.  
 Блок импортов представляет из себя ключевое слово import и >= 1 строк с названием модуля, который мы импортируем, они могут быть указаны следующим образом:

import “fmt”

import (“fmt”)

import (“fmt”;”os”)

import(

“fmt”

“os”

)

Как можно увидеть из этого примера в golang практически всегда символ новой строки и точка с запятой равнозначны.

Объявления структур имеют следующую структуру – ключевое слово type, имя структуры, ключевое слово struct, и блок с полями структуры, каждое поле представляет из себя имя поля, тип и опционально тегом в кавычках, например:

type StructName struct {

age int

name, surname string

uuid string `unique identifier`

}

Объявления интерфейсов очень похожи на объявления структур – сначала идет ключевое слово type, затем имя интерфейса, ключевое слово interface и объявления методов, например:

type geometry interface {

area() float64

perim() float64

}

В golang довольно своеобразно реализовано ООП и в языке не существует ключевых слов для указание реализации интерфейса и наследование классов. Сами разработчики описывают подход используемый в golang так: если что-то выглядит как утка, ведет себя как утка, то скорее всего это утка (чтобы реализовать интерфейс нужно чтобы структура реализовала все методы из интерфейса).

Также, вне функций и методов могу находиться объявления глобальных переменных и констант, например:

var a uint64 = 12

const b string = “Some text”

Перейдем теперь к функциям и методам. В начале обоих этих сущностей находится ключевое слово func, затем указана структура, если это метод, название функции или метода, аргументы и возвращаемое значение(-я). Например, метод может выглядеть так:

|  |  |
| --- | --- |
|  | func (r \*rect) area() int {  return r.width \* r.height  }  А функция так:  func plus(a int, b int) int {  return a + b  }  Или так:  func vals() (int, int) {  return 3, 7  }  Давайте рассмотрим, какие сущности могут находиться в теле функции или метода, мы разделили их на следующие возможные блоки:   1. Вызов функции 2. Возвращение значения или какой-то переменной/функции/метода 3. Присваивание (создание новой переменной или изменение старой) 4. Блок if/else if/else 5. Циклы 6. Switch case 7. Сущности связанные с многопоточностью в Golang – горутины, defer функции, select блок. |
|  |  |

Рассмотрим все эти сущности более подробно. Начнем с вызова функции, которая в нашем анализаторе называется inline\_call\_block и представляет себя имя функции с аргументами, именем функции может выступать поле структуры с неограниченным уровнем вложенности, например:

someFunc(a, b)

someStruct.someFunc()

someStruct.anotherStruct.someFunc(a)

Возвращаемое значение представляет из себя следующую конструкцию – ключевое слово return и какое-то выражение или выражения разделенные запятой, например:

return 2+2

return someValue

return someFunc()

return 2+2, value

return someValue, someStruct.someFunc()

Присваивание — это выражение, которое может быть написано несколькими свойствами, благодаря синтаксису golang. Это может быть как обычное присваивание переменной/константы c указанием типа: val a string = “Some string”, так и неявное приведение типов: a := “Some string”, при таком типе присваивания golang автоматически определяет что указанное значение является строкой.

Ветвления в golang реализовано с помощью аналогичных для других высокоуровневых языков конструкций виде if/else if/else. Единственное отличие от других языков заключается в том, что в golang не нужно использовать скобки, чтобы ограничить условия. Конструкция if может выглядеть следующим образом:

if 7%2 == 0 {

fmt.Println("7 is even")

} else {

fmt.Println("7 is odd")

}

if num := 9; num < 0 {

fmt.Println(num, "is negative")

} else if num < 10 {

fmt.Println(num, "has 1 digit")

} else {

fmt.Println(num, "has multiple digits")

}

Давайте рассмотрим циклы. В отличие от прочих высокоуровневых языков в golang используется одна единственная конструкция для обозначения циклов – for. Поэтому в условии цикла for не обязательно должно быть три условия разделенных точкой с запятой. В остальном цикл стандартный для подобных языков.

for i <= 3 {

fmt.Println(i)

i = i + 1

}

for j := 0; j < 3; j++ {

fmt.Println(j)

}

for i := range 3 {

fmt.Println("range", i)

}

for {

fmt.Println("loop")

break

}

Конструкция switch case также практически не отличается от других Си подобных языков, из отличий можно выделить только отсутствие скобок и возможность указывать в качестве условия case выражения с условием, а не только с константным значением, например:

switch i {

case 1:

fmt.Println("one")

case 2:

fmt.Println("two")

case 3:

fmt.Println("three")

}

switch time.Now().Weekday() {

case time.Saturday, time.Sunday:

fmt.Println("It's the weekend")

default:

fmt.Println("It's a weekday")

}

switch {

case t.Hour() < 12:

fmt.Println("It's before noon")

default:

fmt.Println("It's after noon")

}

Давайте рассмотрим синтактические конструкции для работы с многопоточностью, ради которых golang и создавался.

Для начала рассмотрим горутины – функции, выполняемые в отдельных потоках. От обычных методов их отличает лишь наличие ключевого слова go перед вызовом. Также функции могут быть анонимными – указанными непосредственно внутри другой функции без предварительного объявления, аналог лямбда функций из других языков (например javascript).

go f("goroutine")

go func(msg string) {

fmt.Println(msg)

}("going")

Далее, из конструкций относящихся к многопоточности у нас есть defer функции. *Defer* используется для обеспечения того, чтобы вызов функции был выполняется позже при выполнении программы, обычно для целей очистки или закрытия файлов. Defer часто используется там, где, в других языках будет использоваться блок finally.

f := createFile("/tmp/defer.txt")

defer closeFile(f)

writeFile(f)

В данном примере функция closeFile будет выполнена после того, как функция writeFile завершится.

Перед тем как мы перейдем к рассмотрению select блоков, необходимо разобрать такое понятие как канал в golang. На самом деле оно уже использовалось ранее, например в присвоении, но логически правильнее будет разобрать его именно в блоке многопоточности. В языке программирования Go каналы (channels) — это мощный средство синхронизации и взаимодействия между горутинами (легковесными потоками). Каналы предоставляют способ безопасной передачи значений между горутинами без необходимости использования других синхронизирующих примитивов, таких как мьютексы.

Основные функции каналов:

1. Коммуникация: Каналы позволяют горутинам обмениваться данными. Причём это однонаправленная или двунаправленная передача данных, зависящая от того, как канал объявлен (отправка данных, получение данных, или оба).
2. Синхронизация: Каналы могут быть использованы для синхронизации выполнения нескольких горутин. Например, можно использовать канал для ожидания завершения работы нескольких горутин перед продолжением выполнения основной программы.
3. Управление ресурсами: Каналы могут использоваться для контроля доступа к ресурсам. Например, создание канала с ограниченной вместимостью может помочь регулировать количество одновременных доступов к ресурсу.

Типы каналов:

* Не буферизованные каналы: передача элемента блокирует отправляющую горутину до тех пор, пока другая горутина не прочитает элемент из канала. Это обеспечивает сильную синхронизацию между горутинами.
* Буферизованные каналы: позволяют отправлять несколько элементов до блокировки отправителя, если буфер полон. Получатель может читать из канала без блокировки до тех пор, пока буфер не опустеет.

func main() {

ch := make(chan int) // Создаем канал для передачи данных типа int

go func() {

println(“We are in gorutine!”)

ch <- 123 // Отправляем значение в канал

}()

val := <-ch // Получаем значение из канала

fmt.Println(val) // Вывод: 123

}

Конкретно в этом примере, если бы мы не использовали каналы, содержимое горутины могло бы вообще не выполнится, так как передача выполняется с помощью golang runtime и он не гарантирует что дочерние потоки будут выполнятся после завершения главного потока.

Таким образом, каналы в Go — это фундаментальный инструмент для построения конкурентных и параллельных программ, облегчающий реализацию сложных сценариев с множеством взаимодействующих компонентов.

Давайте разберем последнюю конструкцию относящуюся к многопоточности в go – select. В языке программирования Go, оператор select позволяет одновременно ожидать множество операций ввода-вывода через каналы, делая его особенно полезным для управления несколькими каналами в горутинах. Это уникальное средство Go для обработки конкурентности, которое помогает эффективно и просто управлять несколькими каналами. Синтактически select очень похож на switch case, однако это две абсолютно разные конструкции.

Основные особенности и использование select:

1. Множественный выбор: select позволяет горутине ожидать множественные операции с каналами (как отправку, так и получение) и выполнять блок кода в зависимости от того, какая операция может быть выполнена первой.
2. Неблокирующий ввод/вывод: С помощью select можно реализовать неблокирующие операции чтения или записи, используя default случай, который выполняется, если ни один из каналов не готов для выполнения операции.
3. Синхронизация и тайм-ауты: Очень удобно использовать select с таймерами и тикерами для реализации тайм-аутов и регулярных операций без блокировки основной горутины.

Пример кода с select:

func main() {

ch1 := make(chan string)

ch2 := make(chan string)

go func() {

time.Sleep(time.Second \* 1)

ch1 <- "completed ch1"

}()

go func() {

time.Sleep(time.Second \* 2)

ch2 <- "completed ch2"

}()

for i := 0; i < 2; i++ {

select {

case msg1 := <-ch1:

fmt.Println("Received:", msg1)

case msg2 := <-ch2:

fmt.Println("Received:", msg2)

case <-time.After(time.Second \* 3):

fmt.Println("timeout")

}

}

}

В этом примере select используется для обработки сообщений из двух каналов. Он ожидает сообщения от двух горутин, которые отправляют данные в разное время. Кроме того, в select включен тайм-аут, который срабатывает, если ни одно из сообщений не получено в течение заданного времени. Это помогает избежать бесконечного ожидания, если один из каналов не отправляет данные.

Таким образом, select в Go — это мощный инструмент для управления множественными каналами и тайм-аутами, что делает его незаменимым при написании конкурентных приложений.

## **Тестирование**

Чтобы протестировать разработанную грамматику языка, мы решили запустить нашу программу, на реальном проекте graceful с github. Разберем с какими проблемами мы столкнулись.

* 1. Использование функций и каналов в качестве полей структуры.

Изначально в полях наших структур не подразумевалось использование функций и каналов в качестве полей структуры, однако из тестового проекта стало понятно, что такое в теории может случиться, поэтому мы расширили сущность, отвечающую за поля структуры, в результате чего у появилась возможность описывать структуры следующим образом:

type Server struct {

ConnState func(net.Conn, http.ConnState)

LogFunc func(format string, args ...interface{})

interrupt chan os.Signal

}

* 1. Создание структур

Также при тестировании приложения у нас возникли проблемы с созданием структур, как оказалось, при создании структуры, в нее может передаваться другая функция создания другой структуры. В результате внесенных изменений наш анализатор начал поддерживать такие конструкции:

srv := &Server{

Timeout: timeout,

TCPKeepAlive: 3 \* time.Minute,

Server: &http.Server{Addr: addr, Handler: n},

Logger: DefaultLogger(),

}

* 1. Передача пустого интерфейса в качестве аргумента функции/метода.

Передача пустого интерфейса (interface{}) в качестве аргумента метода в Go используется для создания функций или методов, которые могут принимать аргументы любого типа. Пустой интерфейс не определяет никаких методов, поэтому любой тип данных удовлетворяет ему. Это делает его очень гибким, позволяя функциям или методам работать с данными, которые могут быть совершенно разными по своей структуре или природе.

Например, стандартные функции вывода имеют пустой интерфейс, в качестве последнего аргумента.

В результате мы учли, что в качестве аргумента методов можно передавать интерфейсы, и теперь у нас могут использоваться следующие конструкции:

func printAnything(v interface{})

func (srv \*Server) logf(format string, args ...interface{})

* 1. Использование двумерных массив и карт (map)

В языке программирования Go, многомерные карты (или вложенные карты) — это структуры данных, где каждый ключ карты ассоциируется с другой картой в качестве значения. Это позволяет создавать более сложные структуры данных, которые можно использовать для представления данных с несколькими уровнями ключей, например, для хранения таблиц или матриц, где доступ к элементу осуществляется по нескольким ключам.

В результате реализации этих структур мы можем использовать подобные структуры

matrix := make(map[string]map[string]int)

* 1. Передача пустой структуры в качестве аргумента или значения

Использование пустой структуры struct{} в языке программирования Go имеет несколько особенностей и применений. Пустая структура не содержит никаких полей и, соответственно, не использует память. Это делает её идеальным выбором для определённых сценариев использования, когда нужно обозначить наличие ключа в карте или передать сигнал без передачи конкретных данных.

Одним из самых распространённых применений пустой структуры является использование в качестве типа значения в картах для создания множества (set-like structure). Это особенно полезно, когда важно только наличие ключа, а значение не несёт полезной информации. Например, если вам нужно отслеживать, какие пользователи активны, можно использовать карту с ключами, представляющими пользователей, и значениями типа struct{}:

activeUsers := make(map[string]struct{})

# **ВЫВОД**

В ходе выполнения работы нами была изучена грамматика golang. Мы разработали парсер этого языка используя такие технологии как bison и flex. Так как golang является весьма многофункциональным языком с большим количеством конструкций разработка была довольно затруднительной. Грамматика языка разрабатывалась с использованием дедуктивного подхода, когда сначала описывались общие конструкции языка, а затем «до реализовывались» частные случаи.

Также в ходе разработки языка произошли конфликтные ситуации в результате чего парсер не всегда работал корректно, однако, у нас получилось решить и эту задачу.

# **ПРИЛОЖЕНИЕ**

## **Flex**

%{

#include <stdio.h>

#include "yacc.tab.h"

void yyerror(char \*str);

void error(char s){

printf("Error in line: %d\n", yylineno);

exit(1);

}

%}

%option yylineno

%option noyywrap

%%

"package" return Package;

"import" return Import;

"func" return Func;

"return" return Return;

"go" return Go;

"chan" return Chan;

"select" return Select;

"defer" return Defer;

"const" return Const;

"var" return Var;

"map" return Map;

"make" return Make;

"type" return Type;

"struct" return Struct;

"interface" return Interface;

"if" return If;

"else" return Else;

"case" return Case;

"switch" return Switch;

"default" return Default;

"fallthrough" return Fallthrough;

"continue" return Continue;

"break" return Break;

"range" return Range;

"for" return For;

\= return Equals;

\: return Colon;

\+ return Plus;

\- return Minus;

\\* return Times;

\/ return Devides;

\% return Mod;

\& return And;

"||" return StatementOr;

"&&" return StatementAnd;

"==" return StatementEquals;

"!=" return StatementNotEquals;

":=" return DynamicAssignment;

"..." return Ellipsis;

\| return Or;

\, return Comma;

\^ return Xor;

\! return Not;

\( return OpenBracket;

\) return ClosingBracket;

\{ return OpenCurlyBracket;

\} return ClosingCurlyBracket;

\[ return OpenSquareBracket;

\] return ClosingSquareBracket;

"<-" return ChannelArrow;

\< return Less;

\> return More;

[[:alpha:]\_][[:alnum:]\_]\* return Variable;

((0)|([1-9][0-9]\*(\_?[0-9]+)\*))([eE][-+]?[0-9]+(\_?[0-9]+)\*)? return DecInt;

(0[oO]?(\_?[0-7]+)\*)([eE][-+]?[0-9]+(\_?[0-9]+)\*)? return OctInt;

0[bB](\_?[01]+)\* return BinInt;

(0[xX](\_?[0-9a-fA-F]+)+)([pP][-+]?[0-9]+(\_?[0-9]+)\*)? return HexInt;

(([0-9]+(\_?[0-9]+)\*\.[0-9]\*)|([0-9]\*\.[0-9]+(\_?[0-9]+)\*))([eE][-+]?[0-9]+(\_?[0-9]+)\*)? return DecFloat;

((0[xX](\_?[0-9a-fA-F]+)+\.[0-9a-fA-F]\*)|(0[xX][0-9a-fA-F]\*\.(\_?[0-9a-fA-F]+)+))([pP][-+]?[0-9]+(\_?[0-9]+)\*) return HexFloat;

\"(\\.|[^"\\])\*\" return String;

\`(\\.|[^`\\])\*\` return String;

["';.,~] return yytext[0];

"//".\* ;

[/][\*][^\*]\*[\*]+([^\*/][^\*]\*[\*]+)\*[/] ;

[[:space:]]+ ;

. error(\*yytext);

%%

## **Bison**

%{

#include <stdio.h>

#include <string.h>

extern int yylineno;

extern int yylex();

int error\_flag = 0;

void yyerror(char \*str){

printf("Error in line: %d\n", yylineno);

error\_flag = 1;

}

%}

%nonassoc IFX

%start program\_struct

%token Package

%token Import

%token Func

%token Return

%token Go

%token Defer

%token Chan

%token Select

%token Const

%token Var

%token Map

%token Make

%token Type

%token Struct

%token Interface

%token If

%token Else

%token Case

%token Switch

%token Default

%token Fallthrough

%token Continue

%token Break

%token Range

%token Do

%token While

%token For

%token DecInt

%token OctInt

%token BinInt

%token HexInt

%token DecFloat

%token HexFloat

%token String

%token Variable

%token Plus

%token Equals

%token Colon

%token Minus

%token Times

%token Devides

%token Mod

%token And

%token Or

%token StatementOr

%token StatementAnd;

%token StatementEquals;

%token Xor

%token Not

%token Comma

%token OpenBracket

%token ClosingBracket

%token OpenCurlyBracket

%token ClosingCurlyBracket

%token OpenSquareBracket

%token ClosingSquareBracket

%token ChannelArrow

%token Less

%token More

%token StatementNotEquals

%token DynamicAssignment

%token Ellipsis

%left Plus Minus Times Devides

%right UNARY

%%

program\_struct : package\_block import\_block program

;

program : default\_block

| program default\_block

;

default\_block : type\_block

| assignment

| Func func\_block

;

type\_block : Type Variable Struct OpenCurlyBracket ClosingCurlyBracket

| Type Variable Struct OpenCurlyBracket struct\_fields\_list ClosingCurlyBracket

| Type Variable Interface OpenCurlyBracket ClosingCurlyBracket

| Type Variable Interface OpenCurlyBracket int\_body ClosingCurlyBracket

;

struct\_fields\_list : struct\_field

| struct\_fields\_list struct\_field

;

struct\_field : object\_field

| object\_field Func OpenBracket ClosingBracket

| object\_field Func OpenBracket ClosingBracket object\_field

| object\_field Func OpenBracket args ClosingBracket

| Variable Chan object\_field

| Variable Chan Interface OpenCurlyBracket ClosingCurlyBracket

| Variable Chan Struct OpenCurlyBracket ClosingCurlyBracket

| Map array\_index Interface OpenCurlyBracket ClosingCurlyBracket

| Map array\_index Struct OpenCurlyBracket ClosingCurlyBracket

;

int\_body : func\_head

| int\_body func\_head

;

package\_block : Package Variable

;

import\_block : Import factor

| Import OpenBracket import\_list ClosingBracket

;

import\_list : factor

| import\_list ';' factor

| import\_list factor

;

func\_block : func\_head OpenCurlyBracket func\_body ClosingCurlyBracket

;

func\_head : Variable OpenBracket ClosingBracket

| Variable OpenBracket args ClosingBracket

| Variable OpenBracket ClosingBracket func\_return\_block

| Variable OpenBracket args ClosingBracket func\_return\_block

| Variable OpenBracket ClosingBracket ChannelArrow Chan Struct OpenCurlyBracket ClosingCurlyBracket

| OpenBracket args ClosingBracket func\_head

;

func\_return\_block : object\_field

| Chan object\_field

| OpenBracket func\_return\_block Comma object\_field ClosingBracket

;

func\_body : code\_block

| func\_body code\_block

;

code\_block : inline\_call\_block

| return\_block

| assignment

| if\_block

| for\_block

| Break

| Continue

| switch\_block

| gorutine\_block

| defer\_block

| select\_block

;

variable\_list : expression

| variable\_list Comma expression

;

return\_block : Return

| Return variable\_list

;

inline\_call\_block : object\_field OpenBracket ClosingBracket

| object\_field OpenBracket args ClosingBracket

| object\_field OpenBracket String ClosingBracket

| object\_field OpenBracket String Comma variable\_list ClosingBracket

;

args : arg

| args Comma arg

;

arg : object\_field

| factor

| object\_field object\_field

| object\_field Chan object\_field

| object\_field Chan Struct OpenCurlyBracket ClosingCurlyBracket

| object\_field Chan Chan Struct OpenCurlyBracket ClosingCurlyBracket

| object\_field Ellipsis object\_field

| object\_field Ellipsis

| object\_field Ellipsis object\_field OpenCurlyBracket ClosingCurlyBracket

| object\_field Ellipsis Interface

| object\_field Ellipsis Interface OpenCurlyBracket ClosingCurlyBracket

;

if\_block : If condition OpenCurlyBracket func\_body ClosingCurlyBracket

| If condition OpenCurlyBracket func\_body ClosingCurlyBracket else\_block

| If condition OpenCurlyBracket func\_body ClosingCurlyBracket if\_else\_block

| If condition OpenCurlyBracket func\_body ClosingCurlyBracket if\_else\_block else\_block

| If object\_field OpenCurlyBracket func\_body ClosingCurlyBracket

| If object\_field OpenCurlyBracket func\_body ClosingCurlyBracket else\_block

| If object\_field OpenCurlyBracket func\_body ClosingCurlyBracket if\_else\_block

| If object\_field OpenCurlyBracket func\_body ClosingCurlyBracket if\_else\_block else\_block

;

if\_else\_block : Else If condition OpenCurlyBracket func\_body ClosingCurlyBracket

| if\_else\_block Else If condition OpenCurlyBracket func\_body ClosingCurlyBracket

;

else\_block : Else OpenCurlyBracket func\_body ClosingCurlyBracket

;

for\_block : For condition OpenCurlyBracket func\_body ClosingCurlyBracket

| For multi\_switch\_condition\_body OpenCurlyBracket func\_body ClosingCurlyBracket

| For OpenCurlyBracket func\_body ClosingCurlyBracket

;

condition : expression

| auto\_type\_assignment ';' condition

| expression StatementAnd condition

| expression StatementOr condition

| expression StatementEquals factor

| expression StatementNotEquals factor

| expression StatementEquals factor StatementAnd condition

| expression StatementEquals factor StatementOr condition

| expression StatementNotEquals factor StatementAnd condition

| expression StatementNotEquals factor StatementOr condition

| expression More condition

| expression Less condition

| expression More Equals condition

| expression Less Equals condition

| Not factor

| OpenBracket condition ClosingBracket

;

channel\_assignment : Variable ChannelArrow expression

| Var Variable Variable Equals ChannelArrow Variable

| ChannelArrow expression

| ChannelArrow inline\_call\_block

| Variable ChannelArrow Struct OpenCurlyBracket ClosingCurlyBracket OpenCurlyBracket ClosingCurlyBracket

;

assignment : auto\_type\_assignment

| channel\_assignment

| object\_field Equals expression

| Var Variable Variable

| Var Variable Variable Equals expression

| Const Variable Variable Equals expression

| Const Variable Equals expression

| Var Variable array\_index Variable

| Var Variable array\_index Variable Equals array\_assignment

| Var Variable array\_index Variable Equals object\_field

| Var Variable array\_index Variable Equals inline\_call\_block

| Var Variable Chan Struct OpenCurlyBracket ClosingCurlyBracket

| object\_field array\_index Equals expression

| object\_field array\_index Equals Struct OpenCurlyBracket ClosingCurlyBracket OpenCurlyBracket ClosingCurlyBracket

| Var Variable Variable Equals Variable array\_index

| object\_field Equals Make OpenBracket array\_index object\_field Comma factor ClosingBracket

| object\_field Equals Make OpenBracket array\_index object\_field Comma factor Comma factor ClosingBracket

| object\_field Equals Make OpenBracket Chan Struct OpenCurlyBracket ClosingCurlyBracket ClosingBracket

| object\_field Equals Make OpenBracket Chan object\_field Comma factor ClosingBracket

| object\_field Equals Func OpenBracket args ClosingBracket OpenCurlyBracket func\_body ClosingCurlyBracket

| object\_field Equals Map array\_index Struct OpenCurlyBracket ClosingCurlyBracket OpenCurlyBracket ClosingCurlyBracket

| variable\_list Equals variable\_list

| object\_field Equals Range object\_field

| object\_field Equals Variable OpenCurlyBracket variable\_list ClosingCurlyBracket

| Variable Plus Plus

| Variable Minus Minus

;

array\_index : OpenSquareBracket expression ClosingSquareBracket

| OpenSquareBracket Ellipsis ClosingSquareBracket

| OpenSquareBracket String ClosingSquareBracket

| OpenSquareBracket ClosingSquareBracket

;

array\_assignment : OpenCurlyBracket factor\_list ClosingCurlyBracket

;

factor\_list : object\_field Colon expression Comma factor\_list

| object\_field Colon expression Comma

| object\_field Colon expression

;

auto\_type\_assignment : variable\_list DynamicAssignment variable\_list

| variable\_list DynamicAssignment Range object\_field

| variable\_list DynamicAssignment inline\_call\_block

| variable\_list DynamicAssignment ChannelArrow Variable

| variable\_list DynamicAssignment Variable array\_index

| variable\_list DynamicAssignment dif\_assignment\_obj

| variable\_list DynamicAssignment Map array\_index Variable array\_assignment

| variable\_list DynamicAssignment Make OpenBracket Map array\_index Variable ClosingBracket

| variable\_list DynamicAssignment Make OpenBracket Chan object\_field ClosingBracket

| variable\_list DynamicAssignment Make OpenBracket Chan Chan object\_field ClosingBracket

| variable\_list DynamicAssignment Make OpenBracket Chan Struct OpenCurlyBracket ClosingCurlyBracket ClosingBracket

| variable\_list DynamicAssignment Make OpenBracket Chan Chan Struct OpenCurlyBracket ClosingCurlyBracket ClosingBracket

| variable\_list DynamicAssignment Variable array\_assignment

| variable\_list DynamicAssignment object\_field OpenCurlyBracket ClosingCurlyBracket

| variable\_list DynamicAssignment object\_field OpenCurlyBracket factor\_list ClosingCurlyBracket

;

dif\_assignment\_obj : object\_field

| dif\_assignment\_obj array\_index

| dif\_assignment\_obj inline\_call\_block

;

expression : factor

| factor Plus expression

| factor Minus expression

| factor Times expression

| factor Devides expression

| factor Mod expression

| factor And expression

| factor Or expression

| factor Xor expression

| OpenBracket expression ClosingBracket

| object\_field OpenCurlyBracket factor\_list ClosingCurlyBracket

;

factor : object\_field

| object\_field array\_index

| DecInt

| OctInt

| HexInt

| BinInt

| DecFloat

| HexFloat

| String

| inline\_call\_block

| Plus DecInt %prec UNARY

| Plus OctInt %prec UNARY

| Plus HexInt %prec UNARY

| Plus BinInt %prec UNARY

| Plus DecFloat %prec UNARY

| Plus HexFloat %prec UNARY

| Minus DecInt %prec UNARY

| Minus OctInt %prec UNARY

| Minus HexInt %prec UNARY

| Minus BinInt %prec UNARY

| Minus DecFloat %prec UNARY

| Minus HexFloat %prec UNARY

;

object\_field : Variable

| OpenBracket object\_field ClosingBracket

| Times object\_field %prec UNARY

| And object\_field %prec UNARY

| Variable '.' object\_field

| Not object\_field

;

switch\_block : Switch Variable OpenCurlyBracket switch\_body ClosingCurlyBracket

| Switch multi\_switch\_condition\_body OpenCurlyBracket switch\_body ClosingCurlyBracket

| Switch auto\_type\_assignment OpenCurlyBracket switch\_body ClosingCurlyBracket

| Switch OpenCurlyBracket switch\_body ClosingCurlyBracket

;

multi\_switch\_condition\_body : multi\_switch\_condition

| multi\_switch\_condition\_body ';' multi\_switch\_condition

| multi\_switch\_condition\_body Comma multi\_switch\_condition

;

multi\_switch\_condition : condition

| assignment

;

switch\_body : case\_block

| switch\_body case\_block

;

switch\_code\_block : inline\_call\_block

| return\_block

| assignment

| if\_block

| for\_block

| Break

| Continue

| switch\_block

;

case\_block : Case condition Colon OpenCurlyBracket func\_body ClosingCurlyBracket

| Case condition Colon switch\_code\_block case\_block

| Case condition Colon switch\_code\_block

| Case auto\_type\_assignment Colon switch\_code\_block

| Case args Colon switch\_code\_block

| Default Colon OpenCurlyBracket func\_body ClosingCurlyBracket

| Default Colon switch\_code\_block

;

anon\_func\_block : Func OpenBracket args ClosingBracket OpenCurlyBracket func\_body ClosingCurlyBracket OpenBracket variable\_list ClosingBracket

| Func OpenBracket ClosingBracket OpenCurlyBracket func\_body ClosingCurlyBracket OpenBracket ClosingBracket

;

gorutine\_block : Go inline\_call\_block

| Go anon\_func\_block

;

defer\_block : Defer inline\_call\_block

| Defer anon\_func\_block

;

select\_block : Select OpenCurlyBracket select\_body ClosingCurlyBracket

;

select\_body : select\_case\_block

| select\_body select\_case\_block

;

select\_case\_block : Case channel\_assignment Colon OpenCurlyBracket func\_body ClosingCurlyBracket

| Case channel\_assignment Colon switch\_code\_block func\_body

| Case channel\_assignment Colon func\_body

| Case channel\_assignment Colon

| Case auto\_type\_assignment Colon func\_body

| Case Variable Equals channel\_assignment Colon func\_body

| Default Colon OpenCurlyBracket func\_body ClosingCurlyBracket

| Default Colon

;

%%

int main(){

yyparse();

if(error\_flag == 0)

printf("Compiled successful!\n");

return 0;

}