

"Введение в компьютерную науку."

Презентация на тему:
"Обзор современных ЯП: Rust и Go"

Т. М. Сметанин (teamleader) Н. Д. Джоев

Руководитель: И.Ф. Травов

НИЯУ МИФИ

Выбор компьютерной науки и работа в команде(1)

Проект непосредственно связан с языками программирования, потому раскрывает 14 компьютерную науку: **(PL) Programming Languages**

Распределение задач в команде проекта было построено на основе распределения пошаговых действий:

- Сметанин Т.М. - тимлид: подготовка плана реализации проекта(1) и распределение задач по категориям(2), отбор информации и проработка шаблона Rust-части проекта(3), LaTeX оформление проекта с использованием шаблонов из Powerpoint(5)
- Джииоев Н.Д. - отбор информации и проработка шаблона Go-части проекта(3), вёрстка и полная разработка шаблонов обеих частей проекта с использованием иных средств создания электронных презентаций (MS Powerpoint)(4), LaTeX оформление проекта с использованием шаблонов из Powerpoint(5)

Выбор компьютерной науки и работа в команде(2)

Все действия распределены по пронумерованным группам, пока группа не завершена переход к следующей невозможен. Также структура повествования в Rust и Go частях разная - она отражает два подхода к изучению ЯП:

- больший уклон в теорию и разбор того, что находится "под капотом"
- упор в практическую реализацию определённых парадигм на которых работает ЯП, создание различных pet-project-ов для понимания теории на практике, постоянное чтение документации

Язык программирования Rust

Выполнил Сметанин Т.М.

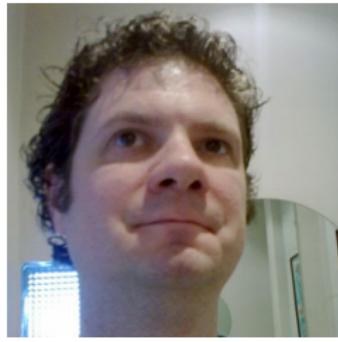
Т. М. Сметанин (teamleader) Н. Д. Джоев
Руководитель: И.Ф. Травов

НИЯУ МИФИ

- В 2006 году Грэйдон Хор начал разработку своего собственного языка названного в честь сорта грибов. В 2009 году Mozilla начала спонсировать его разработки.
- На Mozilla Summit 2010 прошла презентация языка, ещё год шло усовершенствование компилятора и в 2011 произошла его раскрутка.
- В январе 2012 была выпущена официальная альфа-версия *Rust* (0.1).
- В мае 2015 выпущена первая стабильная версия *Rust* (1.0), в которой проведена ревизия интерфейсов и возможностей языка.
- В декабре 2022 *Rust* стал третьим языком который поддерживает в разработке ядра Linux (также поддерживаются *C* и язык ассемблера).

История популярности

- Практически сразу с выходом языка в релиз и переходом в продакшн, он стал одним из самых держащихся на плаву ЯП.
- Многие разработчики отмечали понятность документации, прозрачность языка, открытость и дружелюбность *Rust*-сообщества.
- Первыми компаниями использовавшими этот ЯП в своих проектах стали Mozilla и Dropbox, уже в конце 2015 в Firefox для Linux часть «внутренностей» была переписана на *Rust*.



moz://a

Особенности и отличия от других языков

Rust предлагает несколько весомых преимуществ выгодно отличающих его от других ЯП:

- **Безопасная работа с памятью** – в *Rust* отсутствует сборщик мусора и применяется специальная система работы с типами.
- **Понятный и единый Packet manager** – не все высокоуровневые ЯП обладают единой системой управления пакетами, в *Rust* стандартизирована единая система – *Cargo*.
- **Отсутствие дремучего legacy-кода** – *Rust* это «молодой язык», в котором нет проектов и репозиториев 15-20 летней давности, которые страшно сломать при изменении в сборке проекта, то есть он лишен одной из главных проблем, например, языка *C++*: эта проблема – язык *C*.

Особенности: безопасная работа с памятью

Сегодня большинство высокоуровневых ЯП предлагают сборщик мусора (**Garbage collector**) для ограничения контроля за управлением памятью, в то время как более низкоуровневые языки предоставляют функции такие как `free()` в *C++*, позволяющие свободно работать с динамической памятью; из *Rust* **GC** исключён на одной из стадий пересборки компилятора. Разработчики пришли к выводу о реализации защищённости памяти без **GC**, а с помощью системы заимствованных указателей, которые называли «ссылкой».



Особенности: безопасная работа с памятью

В *Rust* для достижения безопасности памяти применяется концепция «владения» и «заимствования»:

- По умолчанию каждая переменная является неизменяемой.
- Каждое значение присваивается переменной называемой «владельцем» (“owner”). При выходе из области видимости – память освобождается.
- Для передачи переменной в другую часть программы используется «заимствование» – получение доступа к ссылке в памяти, без захвата самой памяти.



Особенности: безопасная работа с памятью

Приведём примеры работы с памятью и стандартной программы «Hello world!»:

```
let hello = "hi mom";
let mut hello = "hi mom";
{
let my_dog = Pug::new();
//do smthng
drop(my_dog)
}
```

```
fn main() {
log 'hello, world';
}

-----
fn main() {
println!("Hello, world!");
}
```

Особенности: **packet manager** - cargo

В современных ЯП принято использовать так называемый **packet manager** – систему призванную навести порядок в установленных библиотеках, открытых проектах и многом другом. Преимуществами такого подхода являются:

- Простая и понятная установка любых пакетов дополняющих ЯП
- Распутывание всех зависимостей требуемых определёнными пакетами
- Безопасное и полное удаление пакетов и зависимостей
- Вменяемая работа с различными репозиториями
- Оптимизация и структуризация файлов исходного кода в языка с различными системами сборки

В *Rust* существует единый **packet manager** и единая система сборки, это позволяет исключить проблему связанную с разработкой в различных **IDE** и различных операционных системах

Особенности: packet manager - cargo

В *Rust* **packet manager-ом** является система **cargo**. Её использование является более предпочтительным чем использование одного компилятора *rustc*. Рассмотрим создание проекта с помощью этой системы:

```
~#cd ~/work //перейдём в заранее выбранный каталог на ЖД  
~#cargo new hello //после выполнения этой команды будет создан каталог hello содержащий файлы проекта  
~#cd hello //по умолчанию в каталоге будет находиться скрытая папка .git, папка src с исполняемым файлом main.rs, файл .gitignore, файл Cargo.toml являющийся конфигурацией проекта Cargo  
~#cargo build // команда создаст исполняемый файл приложения в папке target/debug  
~#cargo run // команда построит проект и сразу запустит созданный файл
```

Особенности: packet manager - cargo

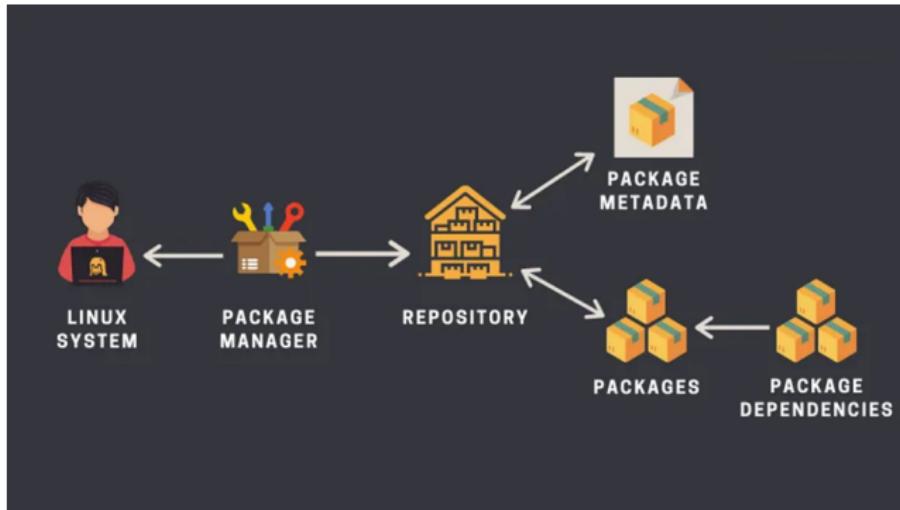
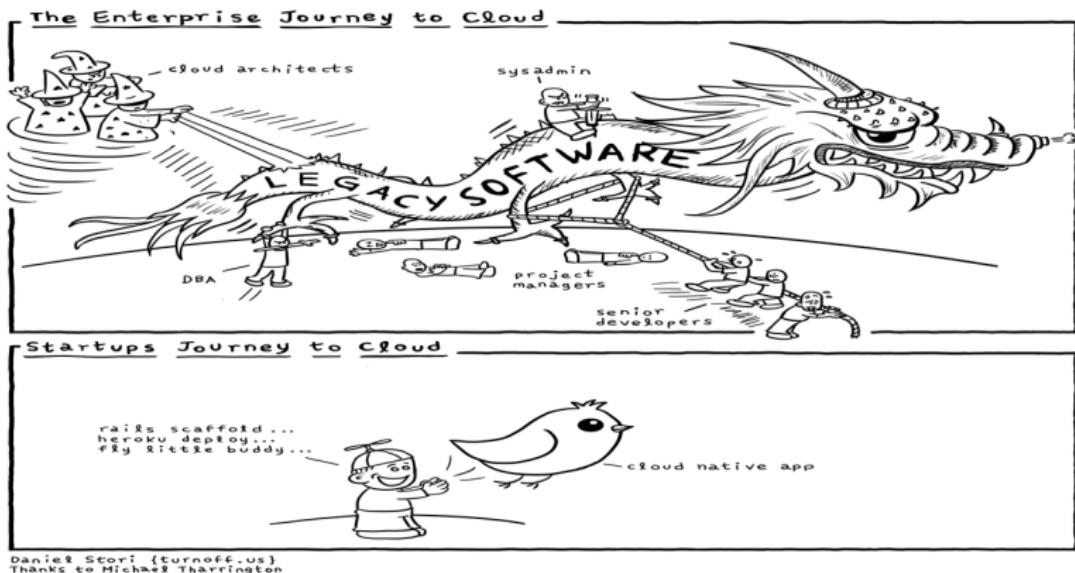


Рис.: Принцип работы менеджера пакетов на основе linux

Особенности: отсутствие legacy-кода

Легаси — это код, про который говорят: «Это ещё Михалыч писал 8 лет назад для синхронизации с сервером, он работает, мы его не трогаем, потому что иначе всё сломается». При этом Михалыча в компании давно нет, документации тоже нет, и проще этот код не трогать совсем.



Особенности: отсутствие legacy-кода

The screenshot shows the PyCharm IDE interface with a project named 'call_Julia [rust_call_julia]'. The file 'main.rs' is open, displaying Rust code that calls a Julia function. The code uses std::process::Command to run 'Julia' with the argument 'main.jl'. It then matches the output of the command to either Ok or Err, printing the result or an error message respectively. The terminal below shows the execution of 'cargo run' and the resulting output 'Hello from Rust' followed by 'Output: hello from Julia function'.

```
Project: call_Julia [rust_call_julia] ~/Documents/PycharmProjects
src
target
  .gitignore
  Cargo.lock
  Cargo.toml
  main.jl
External Libraries
Scratches and Consoles

main.rs
1  #[no_mangle]
2
3  use std::process::Command;
4
5  fn main() {
6      println!("Hello from Rust");
7      let mut cmd : Command = Command::new("Julia");
8      cmd.arg("main.jl");
9      match cmd.output(){
10          Ok(o) => {
11              unsafe{
12                  println!("Output: {}", String::from_utf8_unchecked(o.bytes()));
13              }
14          },
15          Err(e) => {
16              println!("There was an error {}", e);
17          }
18      }
19  }

Terminal
+ Hasan's-Air:call_Julia hasan$ cargo run
+ Finished dev [unoptimized + debuginfo] target(s) in 0.11s
✖ Running `target/debug/rust_call_julia`
Hello from Rust
Output: hello from Julia function
```

Рис.: Скриншот из статьи датируемой 2016 годом, понятно, что код делает совсем простые вещи, но за это время он не потерял свою читаемость и понятность в современной версии языка.

Особенности: препроцессор заменён макросами

```
|system] macbook-pro in ~/dev/cracking-the-coding-interview-rust
# [master x] + cargo build
  Compiling cracking-the-coding-interview v0.1.0 (/Users/brenden/dev/cracking-the-coding-interview-rust)
error[E0609]: no field 'next' on type 'std::rc::Rc<std::cell::RefCell<Node<T>>
'
→ src/bin/c02p01.rs:26:33
26 |         if let Some(cur) = node.next.as_ref().cloned() {
|             ^^^^^^
error: aborting due to previous error

For more information about this error, try 'rustc --explain E0609'.
error: Could not compile 'cracking-the-coding-interview'.

To learn more, run the command again with --verbose.

|system] macbook-pro in ~/dev/cracking-the-coding-interview-rust
# [master x] +
```

Рис.: Читаемость ошибок и быстрое предоставление справки по ним в *Rust*

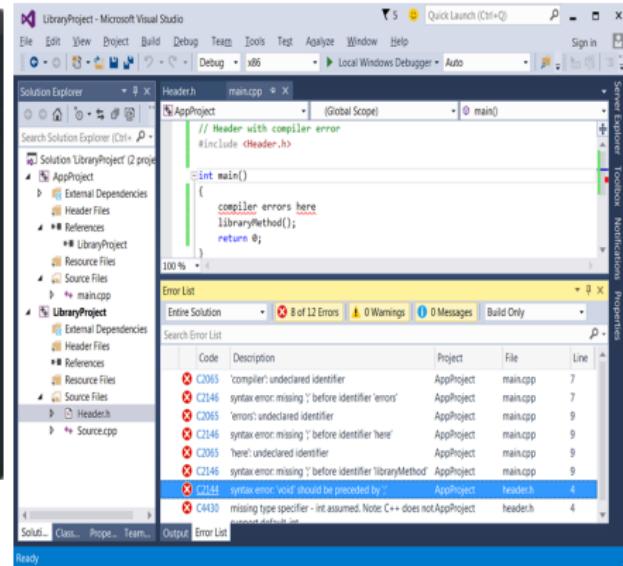


Рис.: *C++*, ну тут всё понятно

Обзор по недостаткам

Как и многие программные продукты построенные на нескольких основных парадигмах ЯП *Rust* имеет ряд недостатков присущих использованию этих конкретных парадигм. Так, можно выделить следующие недостатки:

- Не ко всему программированию можно приложить системный подход, являющийся одной из главных директив *Rust*.
- В языке большое количество сложных конструкций, требуется хорошее знание теоретических основ ЯП, сложно начать изучать *Rust* как первый язык.
- ЯП сравнительно молод и быстро развивается, требуется постоянная слежка за трендами и обучение совершенно новым, не всегда оптимизированным вещам.

Недостатки: системное программирование

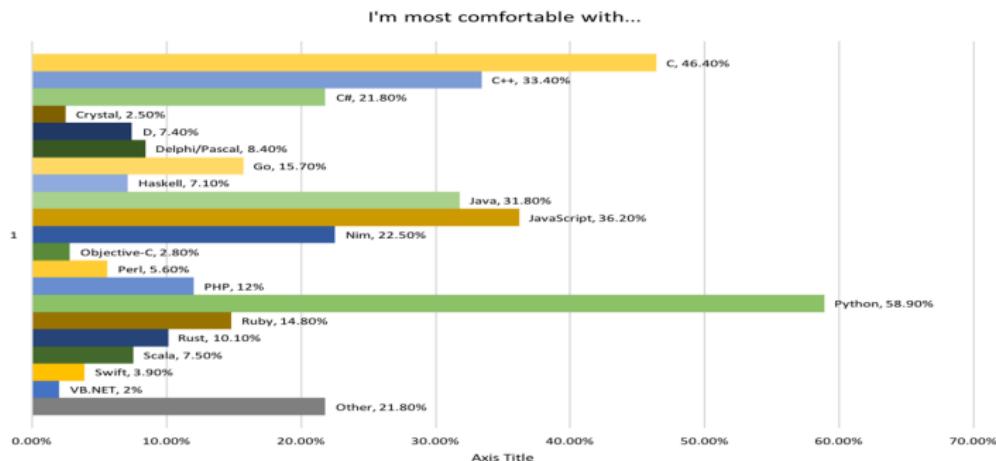
Rust предоставляет прекрасный контроль за использованием памяти, высокий уровень безопасности, высокую скорость исполнения и компиляции файла; это выгодно для масштабных, долгих в разработке, приспособленных под большую команду проектах, но для возможности использования такого языка от программиста требуются высокий уровень знаний и концентрации.



Рис.: Обзорная схема системного подхода к программированию

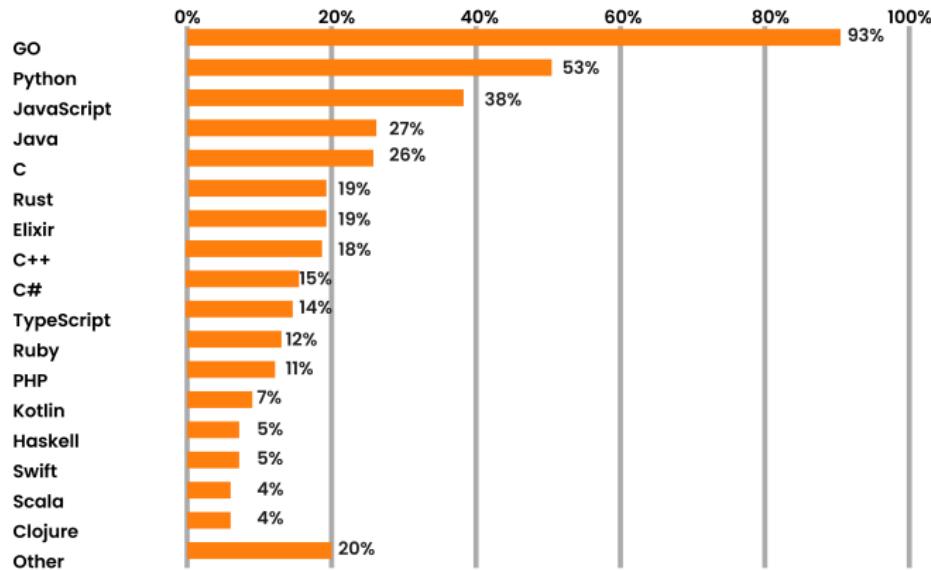
Недостатки: высокий порог вхождения

Вместе с преимуществом, высокая безопасность и глубокий контроль памяти в Rust, это также является и его недостатком: новичок вряд ли будет использовать эти достоинства и извлечь пользу из их применения, тогда ему будет сложно разобраться в тонкостях их применения и механизме работы. Получается сам подход к созданию языка и его работе требует от программиста на нём некоторых знаний о ЯП в целом и опыта работы на других языках.



Недостатки: молодость и незрелость языка

Язык активно улучшается и дорабатывается, в нём реализуются многие значимые и полезные функции, но он ещё не успел накопить большой базы документации, примеров, задач, готовых enterprise проектов для обучения новичков и становления сообщества языка, как у *python*, *java* или *C++*



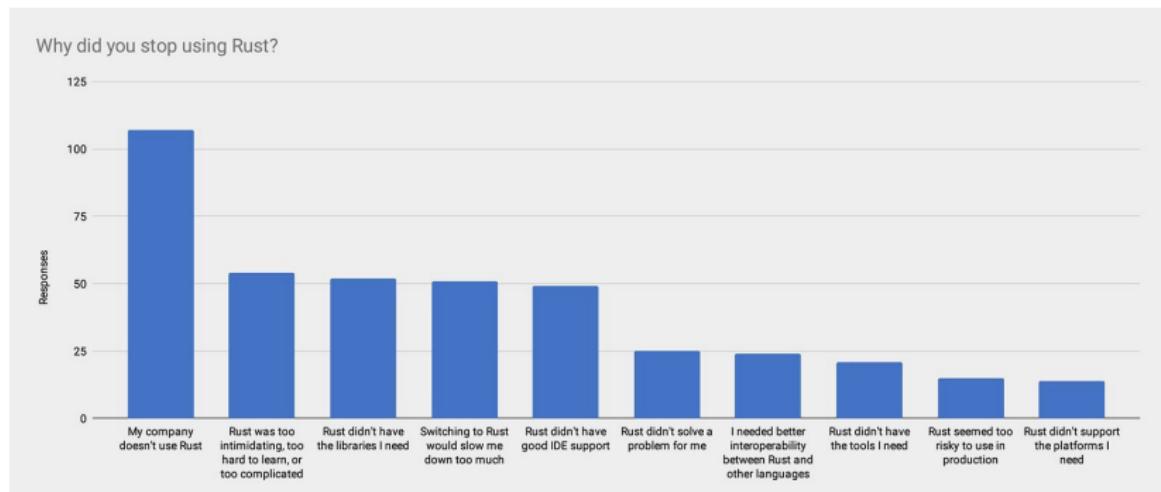
Перспективы: вывод по особенностям

Резюмируя, ЯП *Rust* представляет собой бурно развивающийся компилируемый язык "нового поколения" значительно перерабатывающий концепцию back-end языка и языка системного программирования. За сравнительно недолгое время развития в нём реализованы:

- Действительно безопасная работа с памятью (привет C++)
- Хорошая единая система сборки совмещённая с **packet manager-ом**
- Замена препроцессору в виде макросов, что позволяет обеспечить читаемость ошибок компиляции

Перспективы: вывод по недостаткам

В силу своих особенностей (являющихся в общем-то преимуществами) *Rust* имеет ряд недостатков, которые могут отпугнуть начинающих программистов или программистов с задачами требующими минимальных временных затрат (например большое количество стартапов сейчас реализуется на *Go* так как он позволяет меньше внимания уделять работе с памятью и низкоуровневым моментам в целом).



Бонус - участники конференции RustCon2021 в Москве отвечают на вопрос: "За что я люблю Rust?"

- "В Rust-е? Typesafety относительно плюсов (C++)"
- "Скорость и отсутствие runtime-а"
- "Для меня самое главное - это возможность адекватно работать с ошибками"
- "Для меня в Rust-е самое главное, что он сделан для людей, а не для машины"
- "Безопасность, безопасная разработка"

Язык программирования Go

Выполнил Джииоев Н.Д.

Т. М. Сметанин (teamleader) Н. Д. Джииоев
Руководитель: И.Ф. Травов

НИЯУ МИФИ

Язык программирования Go: История создания

Go (часто также *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, занимавшиеся до этого проектом разработки операционной системы Inferno. Официально язык был представлен в ноябре 2009 года.



Назначение языка Go и его идеология

Язык Go разрабатывался как язык программирования для создания высокоэффективных программ, работающих на современных распределённых системах и многоядерных процессорах. Он может рассматриваться как попытка создать замену языкам C и C++ с учётом изменившихся компьютерных технологий и накопленного опыта разработки крупных систем.



Назначение языка Go и его идеология

По словам Роба Пайка «*Go* был разработан для решения реальных проблем, возникающих при разработке программного обеспечения в Google». В качестве основных таких проблем он называет:

- Медленную сборку программ
- Неконтролируемые зависимости
- Использование разными программистами разных подмножеств языка
- Затруднения с пониманием программ, вызванные неудобочитаемостью кода, плохим документированием и так далее
- Дублирование разработок
- Высокую стоимость обновлений
- Несинхронные обновления при дублировании кода
- Сложность разработки инструментария

Назначение языка Go и его идеология

Основными требованиями к языку стали:

- Ортогональность - язык должен предоставлять небольшое число средств, не повторяющих функциональность друг друга
- Простая и легкоразбираваемая грамматика с минимумом ключевых слов
- Простая работа с типами - типизация должна обеспечивать безопасность, но не превращаться в бюрократию, лишь увеличивающую код
- Отказ от иерархии типов, но с сохранением объектно-ориентированных возможностей
- Отсутствие неявных преобразований
- Сборка мусора

Основные возможности языка

- Go — язык со строгой статической типизацией. Доступен автоматический вывод типов, для пользовательских типов — «утиная типизация»
- Полнценная поддержка указателей, но без возможности применять к ним арифметические операции
- Строковый тип со встроенной поддержкой юникода
- Использование динамических массивов (срезов),хеш-таблиц (словарей), вариант цикла для обхода коллекции
- Средства функционального программирования
- Автоматическое управление памятью со сборщиком мусора
- Средства объектно-ориентированного программирования ограничиваются интерфейсами
- Средства параллельного программирования

Алфавит

Go — регистрозависимый язык с полной поддержкой Юникода в строках и идентификаторах. Идентификатор традиционно может быть любой непустой последовательностью, включающей буквы, цифры и знак подчёркивания, начинающийся с буквы и не совпадающий ни с одним из ключевых слов языка *Go*. При этом под «буквами» понимаются все символы Юникода, относящиеся к категориям «Lu», «Ll», «Lt», «Lm» или «Lo», под «цифрами» — все символы из категории «Nd». Все ключевые слова *Go* пишутся в нижнем регистре. Переменные, начинающиеся с заглавных букв, являются экспортируемыми (`public`), а начинающиеся со строчных букв — неэкспортируемыми (`private`).

Работа с пакетами

Любая программа на Go включает один или несколько пакетов. Пакет, к которому относится файл исходного кода, задаётся описанием package в начале файла. Система пакетов go-среды имеет древовидную структуру, аналогичную дереву каталогов. Для использования в файле кода Go объектов, экспортированных другим пакетом, пакет должен быть импортирован, для чего применяется конструкция import.

```
package main
/* Импорт */
import (
    "fmt"                  // Стандартный пакет для вывода
    "database/sql"         // Импорт вложенного пакета
    w "os"                 // Импорт с псевдонимом
    . "math"                // Импорт без квалификации
    - "gopkg.in/goracle.v2" // Пакет не имеет явного кода
)
```

Система циклов, оператор for

В Go для организации всех видов циклов используется циклическая конструкция `for`:

```
for i < 10 {} // цикл с предусловием, аналог while в С
for i := 0; i < 10; i++ {} // цикл со счётчиком, как for в С
for {} // бесконечный цикл
    // Выход из цикла должен быть организован вручную,
    // используются конструкций return или break
for { // цикл с постусловием
    ... // тело цикла
    if i>=10 {break}
}
for i, v := range arr {} // цикл по коллекции arr
// i - индекс (или ключ) текущего элемента
// v - копия значения текущего элемента массива
```

Оператор множественного выбора switch

Синтаксис оператора `switch` имеет ряд особенностей. Прежде всего, в отличие от *C*, не требуется использование оператора `break`: после отработки выбранной ветви исполнение завершается. Если необходимо продолжение работы, то требуется использовать оператор `fallthrough`:

```
switch value {  
case 1:  
    fmt.Println("One")  
    fallthrough // Далее будет выполнена ветвь "case 0:"  
case 0:  
    fmt.Println("Zero")  
}
```

Особенности архитектуры: обработка ошибок

Язык Go не поддерживает типичного для большинства современных языков синтаксиса структурной обработки исключений. Вместо этого рекомендуется использовать возврат ошибки как одного из результатов функции:

- В последнем параметре функция возвращает объект-ошибку либо пустой указатель `nil`, если функция выполнилась без ошибок.
- Возвращённый функцией объект проверяется и ошибка, если она возникла, обрабатывается.
- Проигнорировать ошибку, возвращаемую из функции.
- При возникновении фатальных ошибок, делающих невозможным дальнейшее выполнение программы, возникает состояние «паники» (`panic`), которое по умолчанию приводит к аварийному завершению программы с выдачей сообщения об ошибке.

Особенности архитектуры: обработка ошибок

```
func ReadFile(srcName string)(result string, err error) {  
    file, err := os.Open(srcName)  
    if err != nil {  
        // Генерация новой ошибки с уточняющим текстом  
        return nil, fmt.Errorf("чтение файла %q: %w", srcName, err)  
    }  
    ...// Дальнейшее исполнение функции, если ошибки не было  
    return result, nil // Возврат результата и пустой ошибки  
}
```

Особенности архитектуры: обработка ошибок

```
func main() {
    defer func() {
        err := recover()
        if v, ok := err.(error); ok {
            fmt.Fprintf(os.Stderr,
                "Error %v \"%s\"\n", err)
        } else if err != nil {panic(err)}
    }()
    a, err := strconv.ParseInt(os.Args[1], 10, 64)
    if err != nil {panic(err)}
    b, err := strconv.ParseInt(os.Args[2], 10, 64)
    if err != nil {panic(err)}
    fmt.Fprintf(os.Stdout, "%d / %d = %d\n", a, b, a/b)
}
```

Особенности архитектуры: обработка ошибок

В примере выше могут произойти ошибки при преобразовании аргументов программы в целые числа функцией `strconv.ParseInt()`. Также возможна паника при обращении к массиву `os.Args` при недостаточном количестве аргументов, либо при делении на нуль, если второй параметр окажется нулевым. При любой ошибочной ситуации генерируется паника, которая обрабатывается в вызове `defer`.

Особенности архитектуры: низкоуровневое программирование

Средства низкоуровневого доступа к памяти сосредоточены в системном пакете `unsafe.Unsafe` предоставляет функции:

- `unsafe.Sizeof()` — аргументом может быть выражение любого типа, функция возвращает реальный размер операнда в байтах, включая неиспользуемую память, которая может появляться в структурах из-за выравнивания
- `unsafe.Alignof()` — аргументом может быть выражение любого типа, функция возвращает размер в байтах, по которому типы операнда выравниваются в памяти
- `unsafe.Offsetof()` — аргументом должно быть поле структуры, функция возвращает смещение в байтах, по которому располагается это поле в структуре
- пакет предоставляет тип `unsafe.Pointer`, в который может быть преобразован любой указатель и указатель любого типа

Особенности архитектуры: низкоуровневое программирование

Описанные преобразования могут быть небезопасны, поэтому их рекомендуют по возможности избегать:

- Во-первых, возможны очевидные проблемы, связанные с ошибочным обращением не к той области памяти
- Более тонким моментом является то, что несмотря на использование пакета `unsafe`, объекты Go продолжают находиться под управлением менеджера памяти и `сборщика мусора`
- Поскольку спецификация Go не даёт точных указаний на то, в какой мере программист может рассчитывать на сохранение актуальности преобразованного в число указателя, существует рекомендация: **сводить подобные преобразования к минимуму**

Реализации

На данный момент существуют два основных компилятора *Go*:

- **gc** — общее название для официального набора инструментов разработки, поддерживаемого группой разработчиков языка. Первоначально он включал компиляторы для **amd64**, для **x86**, для **ARM** и был написан на *C*. В версии 1.5 весь код на *C* был переписан на *Go* и *языке ассемблера*
- **gccgo** — компилятор *Go* с клиентской частью, написанной на *C++*, и рекурсивным парсером.

Также существуют проекты:

- **lugo** — прослойка для компиляции *Go* в **llvm**, написанная на самом *Go*
- **gollvm** — проект компиляции *Go* через систему компиляторов LLVM, развивающийся Google
- **SSA interpreter** — интерпретатор, позволяющий запускать программы на *Go*

Средства разработки

Среда разработки Go содержит несколько инструментов командной строки: утилиту *Go*, обеспечивающий компиляцию, тестирование и управление пакетами, и вспомогательные утилиты [godoc](#) и [gofmt](#). На текущий момент доступны две IDE, изначально ориентированные на язык Go — это проприетарная [GoLand](#) (разрабатывается в JetBrains на платформе [IntelliJ](#)) и свободная [LiteIDE](#) (ранее проект назывался [GoLangIDE](#)). Также Go поддерживается плагинами в универсальных IDE [Eclipse](#), [NetBeans](#), [Komodo](#), [Visual Studio](#), [Zeus](#) и других. Автоподсветка, автодополнение кода на Go и запуск утилит компиляции и обработки кода реализованы в виде плагинов к более чем двум десяткам распространённых текстовых редакторов под различные платформы.

Финиш по обзору современных ЯП

...the End по языкам *Rust* и *Go*, вопросы?

```
вопросы = int(input())
if(вопросы == 0):
    print("Спасибо за ваше внимание!")
else:
    вопросы-=1
```