Современное программирование

Про Rust и не только

```
let title = Title<Article> {
  author:    "Евгений Лепихин",
  started:   date!(2023 - 06 - 30),
  completed: date!(2023 - 09 - 06),
}
```

Об алгебре типов, надежном программировании, метапрограммировании, функциональном подходе, инфраструктуре языков.

Содержание

1 О чем поговорим

В интернете много раз сравнивали Rust с другими языками в контексте безопасности работы с памятью и многопоточностью. В Rust с этим действительно всё хорошо среди языков без рантайма (сборщика мусора). Но широкой общественности почти ничего неизвестно о других достоинствах, которые язык вобрал из дремучего computer science. В результате распространилось мнение, что на безопасности работы с памятью в Rust всё и заканчивается. Это не так.

Темы статьи во многом применимы и к Haskell, Ocaml, Coq, TypeScript, NodeJS и некоторым другим языкам. Задача статьи рассказать, что мир не остановился на циклах с классами, но и не утонул в пучинах математики: посередине есть много применимого на практике.

2 Алгебра типов

2.1 Зачем она?

Чем строже система типов, тем более точно она отвечает на вопрос о корректности (безошибочности) программы на момент компиляции.

Алгебра типов — более удачное решение проблемы, которую пытается решить ООП: код должен описывать сущности из головы программиста. Однако алгебра типов даёт понимание об этих сущностях и компилятору, позволяя ему более полно доказывать корректность программы.

2.2 Теория: произведение типов

Что такое uint8? Это тип, который населен всем множеством возможных значений целых чисел от 0 до 255.

Что такое тип bool? Это тип, который населен двумя возможными значениями: **true** и **false**.

Что такое нижеследующая структура?

```
type Mul struct {
  a: bool
  b: uint8
}
```

Это [именованное] произведение типов bool и uint8. Его можно записать как bool * uint8. Что означает произведение? То, что тип Mul содержит все варианты комбинаций типов bool и uint8, то есть Mul населен $256 \cdot 2 = 512$ возможными значениями.

Небольшая ремарка: вернемся к записи bool * uint8. Это очень напоминает запись типа-кортежа (tuple) в ML-языках, а также похоже на гошную запись (bool, uint8). Совершенно верно: с точки зрения компиляторов, структуры — просто кортежи с именованными элементами.

2.3 Теория: сумма типов

Ознакомились с произведением типов. Что же такое сумма? В традиционных языках сложнее найти аналог, поэтому включим фантазию. Представим, что наша функция возвращает **либо** uint8, **либо** error. Знакомая в Golang конструкция:

```
func GetSpentFunds() (uint8, error) { ... }
```

Казалось бы, проблема решена. Поможет ли нам компилятор избежать багов при таком возвращаемом типе?

```
dollars, err := GetSpentFunds()

// Использовали err, компилятор и линтер довольны
CountOptionalErrorInStatisticsManager(err)
```

```
// Использовали dollars, компилятор и линтер снова довольны
fmt.Printf("user spent %d dollars\n", dollars)
// ... но ой, мы что-то забыли. А компилятор не подсказал.
```

А всё потому, что наша функция вместо суммы типов возвращает их произведение, то есть с точки зрения компилятора у нас всегда есть **и** какое-то число, **и** какая-то ошибка (хотя бы nil).

Было бы круто, если компилятор будет нам указывать, что функция вернула одно значение, но не вернула второе. Изобретем синтаксис, чтобы проиллюстрировать идею:

```
// Возвращаем ЛИБО uint8, ЛИБО error.
func get_some_number() (uint8 || error) {
    if (1 != 1) {
        return errors.New("bug run over your mainboard")
    }

    return 0
}

func main() {
    result := get_some_number()
    // Новая форма switch понимает какие варианты типа могут
    // быть в значении.
    switch result {
        case uint8[[number]]:
            fmt.Printf("got some number: %d\n", number)
        case error[[err]]:
            fmt.Errorf("things happen, got error: %+v", err)
    }
}
```

Чем это лучше? Компилятор теперь знает, что мы возвращаем **один** из двух **вариантов**, и в операторе switch имеет возможность проверить, что мы сравниваем результат со **всеми** возможными **вариантами**, которые может вернуть наша функция, и вернуть ошибку компиляции, если мы забыли обработать вариант error.

Резюмируя: сумма типов A и B населена суммой всех вариантов их значений. Например, тип-сумма uint8 и bool имеет $256+2=258$ значений.

Введение типов-сумм в Golang обсуждают например здесь: https://github.com/golang/go/issues/57644

2.4 Null pointer dereference (не совсем pointer, и не совсем dereference)

Предположим, что нам надо работать с опциональными значениями. Традиционно для этих целей используют указатель, который может ссылаться и на NULL. Это сопряжено с проблемами: нам необходимо перед использованием проверять, что по ссылке объект действительно есть. А что если воспользоваться суммой типов, и компилятор всегда будет знать, что у нас либо есть объект, либо его нет? Тогда он не даст работать со значением как с объектом, если оно может оказаться пустым.

Всё уже придумано за нас. В ML-языках и Rust этот тип называется Option, в Haskell Maybe и т.д. Упрощенно, определение этого типа в Rust выглядит так:

```
enum Option {
    // Вариант с хранением объекта
    Some(obj),
    // Пустой вариант
    None
}
```

Мы никак не сможем напрямую обратиться к obj внутри Some(), не развернув вариантный тип Option. Поэтому при любой попытке использовать obj, у нас он гарантированно будет! И эта гарантия предоставляется компилятором:

```
// функция возвращает контейнер Option, содержащий либо строку -
// в случае успеха чтения, либо None
let user_input = read_line()

// невозможно «достать» user_message из Some/None, не сравнив тип
// с его вариантами
let user_message = Some ... ... // ??? !!!

// но зато можно получить доступ путем извлечения из варианта:
match user_input {
    // Из Some нам досталось user_message, мы можем работать
    // со значением
    Some(user_message) => println!("User wrote: {}", user_message),
    // внутри варианта None ничего нет, ничего сделать не можем.
    // Но обработать вариант в сравнении с образцом мы обязаны,
    // иначе компилятор нам укажет на проблему
    None => ()
}


// В Rust есть более краткая форма сравнения для случаев,
// когда нужно поработать только с одним вариантом:
if let Some(user_message) = user_input {
    println!("User wrote: {}", user_message)
}
```

2.5 Краткое резюме

Алгебраические типы замечательны тем, что с ними можно работать математическими методами. Это своеобразный мостик между программированием, типами данных и доказательством теорем.

Компилятор получает больше понимания о структуре программы и о том, что имел ввиду программист. И чем больше мы конкретизируем типы, тем больше ошибок он нам сможет указать.

2.6 Погружаясь к глубины

2.6.1 Рекурсивные типы

Давайте попробуем описать безопасное в обращении бинарное дерево чисел. В этом разделе опустим некоторые усложенения Rust, связанные с отсутствием сборщика мусора.

Для начала ещё чуть-чуть потренируемся с типами:

```
// Определяем суммирующий тип SumOfTypes
// В нём у нас два варианта. Один контейнер для варианта int, второй
// для строк.
enum SumOfTypes {
    ThisIsIntVariant(int),
    ThisIsStringVariant(string)
}

// Определяем значение типа SumOfTypes
let some_sum_value = ThisIsStringVariant("Hello world")


// Определяем тип-произведение MultiplyOfTypes, обычную структуру
struct MultiplyOfTypes {
    some_int: int;
    some_string: string;
}


// Определяем значение типа MultiplyOfTypes
let some_mult_value = MultiplyOfTypes {
    some_int: 5,
    some_string: "Hello world"
}
```

Итак, дерево. У каждого узла дерева есть значение и два ребенка, где каждый ребенок — это **либо** лист (конечное значение), **либо** поддерево. Как слышим, так и пишем:

```
struct Node {
    value: int,
    left_subtree: NodeVariant,
    right_subtree: NodeVariant
}


// В каждом узле дерева ЛИБО пустое значение (лист дерева),
// ЛИБО узел
enum NodeVariant {
    // Пустой лист
    Leaf,
    // Узел
    Node(node)


// определим дерево со значениями 25, 50, 75
let our_tree =
  Node {
      value: 50;
      left: Node { value: 25, left: Leaf, right: Leaf },
      right: Node { value: 75, left: Leaf, right: Leaf }
```

```
}
```

Определим рекурсивную функцию для печати всех значений из дерева в сортированном порядке (в предположении, что бинарное дерево использовалось по назначению):

```
fn print_tree (tree: NodeVariant) {
    match tree {
        // Встретили лист. Ничего не делаем
        Leaf => return (),

        // Встретили узел
        Node(node) => {
            // Рекурсивно обходим левую ветвь
            print_tree(node.left);
            // Печатаем значение из текущего узла
            print!("{}", node.value);
            // Рекурсивно обходим правую ветвь
            print_tree(node.right)
        }
    }
}
```

Вот так незатейливо и безопасно мы описали бинарные деревья. Теперь компилятор имеет достаточно представления о том, что такое бинарные деревья, и подумает за нас на всех собеседованиях, где мы будем реализовывать распечатку, редактирование или перебалансировку деревьев.

2.6.2 Комбинаторные игры

Нашу фантазию в комбинации типов никто не ограничивает, и мы можем определять довольно сложные сущности, которые по-прежнему будут доступны для валидации компилятором:

```
// Определим тип натуральных чисел
enum NaturalNumber {
    Zero,
    Next(NaturalNumber)
}
// Тут хитрость: у нас есть только Zero и понятие «следующее число».
// Next(Zero) есть единица. Next(Next(Zero)) есть двойка и т.д.


// Опишем вспомогательный тип знака числа. Можно было бы использовать
// bool, но помним: чем подробнее мы описали типы, тем больше
// ошибок нам покажет компилятор.
enum Sign {
    IsPositive,
    IsNegative
}


// Определим тип целых чисел
struct Integer {
    number: NaturalNumber,
    sign_of_number: Sign
```

```
}

// Определим тип рациональных чисел
struct Rational = {
    quotient: Integer,
    fraction: Integer
}
```

Ой, кажется мы изобрели арифметику Пеано! Написать к нашим типам чуть-чуть функций с уже известным вариантом применения оператора match, и сможем с помощью компилятора доказывать математически теоремы!

В качестве экстремального примера дальнейшего углубления: компилятор может выдать ошибку что в функцию, принимающую только простые числа, кто-то пытается передать значение, которое потенциально может быть не простым числом. Rust так не умеет, но например умеет Coq. Гипотетический пример:

```
// Функция принимает тип Integer, но из внутренней логики
// компилятору становится ясно, что это на самом деле должны
// быть простые числа
fn accepts_only_prime_numbers (number: Integer) {
    // некоторый код, из которого следует, что number —
    // простое число
}

let five = 5
// Ошибка!
accepts_only_prime_numbers(five+1)
```

Если заглянуть ещё чуть глубже, то можно прийти к изоморфизму Карри-Ховарда: эквивалентности между математическими доказательствами и типизированными исчислениями, коими и является алгебра типов. И именно на этом построены все идеи тотальной формальной верификации программ, системы доказательства теорем, генераторы формально доказанного кода.

2.6.3 Фантомные типы

Ладно, поиграли с математикой, хочется спуститься на землю. Зачем изобретать свои int'ы, если занимаемся бизнесом?

Говорят, однажды NASA потеряла 50 миллионов, потому что между континентами инженеры не договорились, и прибавили метры к футам. Аварии можно было избежать, если бы они использовали фантомные типы.

И метры, и футы оптимально хранить в числовом типе. Но как объяснить компилятору, что сложить 2 метра и 3 фута не получится, потому что «это другое»? Можно было бы ввести вариантный тип:

```
enum Length {
    Meter(float),
    Foot(float)
```

```
}
```

Но решит ли это проблему? С помощью match будем извлекать число из варианта, вручную каждый раз проверять что прибавляем правильные единицы, а компилятор будет упорно молчать т.к. в итоге мы действительно проверяем все варианты, и числа прибавляем к числам. А код в рантайме наоборот начнет тормозить из-за излишних сравнений.

```
fn sum_length (a: Length, b: Length) -> Length {
    match a {
        // В первом числе метры
        Meter(m1) => {
            match b {
                // Во втором числе метры
                Meter(m2) => return Length::Meter(m1 + m2),
                Foot(_) => panic!("Cannot sum meters and feet!")
            }
        }
        // В первом числе футы
        Foot(f1) => {
            match b {
                Meter(_) => panic!("Cannot sum meters and feet!"),
                // Во втором числе футы
                Foot(f2) => return Length::Foot(f1 + f2)
            }
        }
    }
}
```

Для решения этой проблемы были придуманы фантомные типы — они существуют только в момент проверки кода компилятором. Фантомным типом маркируем основной тип. После проверки корректности типов он исчезает, и в машинный код попадают только инты.

В Golang фантомные типы поддерживаются:

```
type meter int
type foot int

a := meter(42)
b := foot(42)

// Однако ничто не мешает случайно прострелить ногу, ведь любой
// int преобразовывать в любой другой, в языке нет возможности
// ввести ограничение:
sum := a + meter(b)
// (более правильное вычисление):
sum := a + meter(float32(b)*0.3)
```

В Rust тоже есть фантомные типы, однако любое преобразование типов должно быть явно задано программистом. Для сокращения бойлерплейта, специально для этого существуют трейты From и Into, но это за рамками статьи.

2.6.4 Unit: тип с единственным значением

Окей, мы научились суммировать типы, делать бесконечные суммы и произведения. А какими значениями населен пустой кортеж? Можно предположить, раз в нём перемножение нуля типов, то умножая на ноль мы получим ноль вариантов значений. На самом деле, создание экземпляра пустого кортежа порождает тип с одним единственным значением. Таким образом, у пустого кортежа **один** вариант значений (меньше чем у одного бита!), и для вычисления мощности множества значений кортежа правильно использовать формулу:

$$1 \cdot type_1 \cdot type_2 \cdot \ldots \tag{1}$$

Тип unit всегда выражает строго одно значение, «я есть» (или точнее, «меня выразили» или «меня вычислили»). Это нам пригодится в рассуждении о функциях.

2.6.5 Тип функции

Функции тоже имеют типы. Взглянем заново на строчку, с которой начинают школьники:

```
println("Hello world")
```

Ха, легко! Функция принимает string. А возвращает?.. А возвращает она как раз тип unit. Тот самый, который без вариантов либо есть, либо нет. В математике не бывает функций, которые ничего не возвращают, и умные компиляторы активно этим пользуются. Правильный тип функции можно было бы записать так:

```
type hello_world_type: string → unit
```

А какой тип у функции, которая вычисляет число Пи? По аналогии догадаться не сложно:

```
type pi_function_type: unit → float
```

Факультатив: какой тип возвращает функция exit(), ведь она никогда ничего не возвращает? Тут в мире математики случается черная дыра, и в умных компиляторах она обходится разными методами, например введением специального типа «любой тип». Да, результат exit() мы законно можем присвоить в тип любого значения.

2.6.6 Интерфейс как тип

Все знают про интерфейсы. Давайте попробуем о них думать формально в рамках системы типов. Уже известная, что функция имеет тип. Об интерфейсе можно думать как о произведении типов функций, его населяющих. Мысль не очевидная, попробуем на примере:

```
type BoolsAndInts interface {
    GetRandomBool() bool
    GetRandomInt() uint8
}
```

В результате объединения двух функций в интерфейс, у нас появился «тип», с помощью которого можно получать и bool, и int. Всё множество населяющих интерфейс значений есть перемножение множеств значений, которые могут принимать функции. Первая функция может принимать значения

- unit → true
- unit → false

Вторая функция:

- unit → 0
- unit → 1
- unit → 2
- ... (256 вариантов)

Итого, наш интерфейс умеет генерировать $256 \cdot 2 = 512$ вариантов значений. Именно поэтому интерфейс можно считать произведением типов функций.

2.6.7 Как найти площадь Ленина

- Крокодил имеет GetColor(), имеет GetLength(), но не имеет GetWidth()
- Небо имеет GetColor(), но не имеет GetLength() и GetWidth()
- Площадь Ленина имеет GetLength() и GetWidth(), но её цвет определить уже невозможно.

Замечательно, что можно описать интерфейс вида:

```
type CalculatableSquareArea interface {
    GetLength()
    GetWidth()
}
```

и кастить в него любые объекты, для которых возможно посчитать площадь! Но что делать, если для абстракций, с которыми мы планируем работать, ещё не придумали слов в человечьем языке, а интерфейс как-то назвать надо? Нетрудно догадаться, что комбинаций различных функций на практике случается огромное количество. Для каждой комбинации

описывать интерфейс и выжимать из себя новые безумные слова? Даже если интерфейс будет использован в одной единственной функции?..

Не надо заниматься комбинаторикой вручную, компиляторы уже давно всё умеют. Можно описать тип параметра функции: он должен реализовывать несколько нужных нам интерфейсов, например уметь одновременно GetLength() и GetWidth(). И неважно как это называется, такое **произведение интерфейсов** само по себе достаточно описывает необходимую нам абстракцию.

```
fn GetArea(obj: GettableWidth + GettableLength) -> int {
    obj.getWidth() * obj.getLength()
}
```

2.6.8 Еще раз о типе unit

В последнем кусочке кода можно обратить на «забытое» слово return. Тут нет ошибки, функция возвращает значение последнего выражения, в нашем случае obj.getWidth() + obj.getLength(). Любое выражение в языке Rust имеет значение:

```
let next_optional_number = match optional_int {
    // В типе Option содержался None, присваиваем
    // в next_optional_number вариант None
    None => None,
    // В типе Option содержалось какое-то число,
    // увеличим его.
    Some(number) => Some(number + 1)
}
```

В функции print_hello() последним выражением является println!(), который возвращает тип unit. Поэтому сама функция print_hello() тоже возвращает unit:

```
fn print_hello () {
    println!("Hello")
}
```

2.6.9 А был ли мальчик?

Посмотрим подробнее на возможности сравнения с образцом. Предположим, что надо написать калькулятор родственных отношений, и сейчас как раз описываем их типы:

```
enum Relation {
    Grandfather,
    Grandmother,
    Father,
    Mother,
    Daughter
}
```

Дальше написали функцию, которая позволит выводить имя отношения в понятном людям формате:

```
fn get_name (relation: Relation) -> String {
    match relation {
        Grandfather => "дедушка",
        Grandmother => "бабушка",
        Father => "папа",
        Mother => "мама",
        Daughter => "дочь"
    }
}
```

Далеко в другом конце кода функция, возвращающая индекс поколения:

```
fn get_level (relation: Relation) -> int {
    match relation {
        Grandfather => 2,
        Grandmother => 2,
        Father => 1,
        Mother => 1,
        Daughter => 0
    }
}
```

И еще 50 функций раскиданы по мегабайтам кода. И тут наш дядя, всегда всех мучивший напоминаниями о памятных датах, указывает на баг: мы забыли описать в типе вариант Son! Дописываем. Но как же нам теперь найти все места, где сравнения стали не полными? Компилятор сделает это за нас.

Более того, если внутри некоторых вариантов различные структуры, в структурах поля, в полях кортежи, а в кортежах снова вариантные типы, и мы такой развесистый тип сравниваем по образцу — даже здесь компилятор подскажет, что сравнение неполное.

3 Метапрограммирование

Выразительность программы через типы хороша, но решает лишь одну из проблем разработчика. А что если с помощью кода автоматизировать не только конечную цель программы, но и само написание кода? Эту задачу решает метпрограммирование.

В современном виде, препроцессору языка достается готовое Abstract Syntax Tree (AST), по сути ресурсивный тип данных, описывающий код программы, который препроцессор может произвольно менять.

3.1 Удобная передача ошибки вверх по стеку

Всем знакомый бойлерплейт в golang:

```
val, err := FallibleFunction()
if err != nil {
    return err
}
```

Конструкция настолько частая, что хочется какой-нибудь макрос, который каждый раз будет писать её за нас, и добавит-таки исключения в язык Go:

```
val := @throw FallibleFunction()
```

В Rust помимо стандартного типа Option есть подобный стандартный тип Result. Для него придуман макрос, подобный описанному выше. А поверх макроса позже был придуман синтаксический сахар — оператор «?»:

```
let val = fallibleFunction()?

// Всё! Если fallibleFunction() вернула вариант Eror из типа Result,
// то этот Result будет передан выше. Если же функция вернула
// значение, то оно будет присвоено в val.
```

Оффтопик:

Кстати, а почему в качестве ошибки можно возвращать только то, что крякает как error, и нельзя вернуть просто int? Можно. Result — generic тип, ему всё равно какой тип в него складывают в качестве ошибки.

3.2 Генерация справочников

Предположим, мы реализуем локализацию программы. Нам нужны константы с сообщениями на языке по умолчанию, и нужен полный их список. Всё это мы держим в коде, чтобы компилятор нам указывал на опечатки в названиях констант и не давал получить неработающий код:

```
type MsgId int64

const (
```

```
    MsgIdError          MsgId = itoa
    MsgIdDone           MsgId
    MsgIdConfirmation   MsgId
    // И еще 1000 сообщений
)


// Интерфейсу редактирования переводов нужно знать все возможные
// сообщения с их идентификаторами

type TranslationMessage struct {
    id             MsgId
    defaultMessage string
    translations   map[LanguageId]string
}


var (
    MessagesCatalog = []string{
        TranslationMessage{
            id: MsgIdError,
            defaultMessage: "Got error",
            translations: make(map[LanguageId]string, 0)
        },
        TranslationMessage{
            id: MsgIdDone,
            defaultMessage: "Completed successfully",
            translations: make(map[LanguageId]string, 0)
        },
        // ...
    }
)


// Обращаемся к переводу, грубый пример:
fmt.Printf(
    "%s (y/n)",
    MessagesCatalog[MsgIdConfirmation].defaultMessage,
)
```

В глубинах кода теперь компилятор подскажет где мы не правы и указали неправильный MsgId. Но ведение каталога сообщений вызывает сильнейшую боль. Если же доверить его ведение специально обученному макросу, всё снова станет легко:

```
translations!(
    Error: "Got error",
    Done: "Completed successfully",
    Confirmation: "Are you sure?",
)

// Обращаемся к переводу
println!(
    "{} (y/n)",
    messagesCatalog[msgIdConfirmation].defaultMessage,
)
```

3.3 Безопасный printf()

В языках программирования известна проблема printf. По сути, формат строчки — это отдельный язык, который хорошо бы уметь интерпретировать во время компиляции, и в машинный код складывать уже готовый алгоритм печати. Но есть ещё одна проблема: сопоставление типов подстановок с типами передаваемых параметров. Например, %s должен принимать только строки. Методами обычного программирования эта задача неразрешима, и сопоставлять типы придется в рантайме. Однако, если доверить обработку параметров printf() препроцессору, то ничего сложного: препроцессор парсит шаблон сообщения, типизирует ожидаемые параметры и уже такой типизированный код передает компилятору. Именно так устроен класс макросов format!() в Rust. Это макросы write!(), println!() и т.д.

В некоторых языках это решается без метапрограммирования, с помощью Generalized Algebraic Data Types, но это следующий этаж подземелья алгебры типов (и не последний).

3.4 Метапрограммирование типов

Можно обратить внимание, что типы гомогенны: любой тип является деревом сумм и произведений подтипов и базовых типов. А раз они однородны, с ними можно работать обобщенными методами. Макросом можно пробежаться по полям структуры и для каждого поля сгенерировать код вида print(v.field_name) — получится заготовка для форматированной печати структур. В Rust для этого есть отдельный класс derive-макросов.

Можно пойти дальше, и с помощью derive научить программу сериализовать произвольные структуры в JSON и десериализовать обратно. Можно сделать отображение типов в таблички БД, параметры командной строки, генератор документации к формату конфигурационного файла и т.д. Подобные макросы в ассортименте были разработаны сообществом и опубликованы в форме библиотек, что позволяет за 5 минут написать чрезвычайно функциональную заготовку будущей программы: автоматический разбор параметров командой строки в структуры и варианты, парсинг конфига, документация к конфигу и пр.

3.5 В глубинах метапрограммирования

Я бы предположил, что идеальным языком для мета-программирования является Lisp. По сути, в своей основе это даже не язык, а стандарт элементарного синтаксиса, имеющего минимальный набор лексем для описания рекурсивных списков. Синтаксис состоит лишь из операторов группировки элементов (скобочки) и абстракции самих элементов: символы, строчки, циферки, группы. Этого достаточно, чтобы строить произвольные деревья:

```
;; пустая группа (кортеж)
()

;; Не пустая группа из пяти элементов.
;; Сейчас это просто синтаксис, не наделенный смыслом. Но если
;; применить к этой группе один из диалектов Lisp, то появится
;; смысл: инициализация списка (условно, массива) из четырёх
;; элементов.
(list 1 2 3 4)


;; группа из двух элементов "defun" и "hello-world"
(defun hello-world)

;; пример простого дерева
(defun hello-world ()
  (print "Hello world"))
```

Думайте о скобочках, как о сильно упрощенной JSON-нотации. Элементарность этого синтаксиса замечательна тем, что любой желающий может быстро написать парсер Lisp'а. И есть много готовых парсеров, они предоставляют программисту готовый AST — пока ещё не наделенный смыслом. И поскольку в «базовом» Lisp нет ключевых слов, любой желающий легко может создать свой диалект языка: объявить, что элемент defun является ключевым словом определения функции, for ключевым словом для циклов и т.д.

Не надо изобретать отдельный синтаксис для написания макросов: сама программа, изменяющий её макрос и результат работы макроса являются структурами одного и того же типа данных. Этот подход открывает широчайшие возможности для метапрограммирования: тело «макроса» получает на вход кусочек дерева (элементарный AST), который можно модифицировать, создавая языковые абстракции любой сложности. Макросы для макросов, изобретение своего варианта ООП или ФП, интерпретируемые конфиги с циклами и функциями, и что угодно ещё.

4 Упрощаем жизнь дальше

4.1 Цепочки вызовов

Ранее был описан тип Option. Напомню:

```
type Option =
  // Вариант с хранением объекта
  | Some(obj)
  // Пустой вариант
  | None
```

Позже для него была написана функция, возвращающая следующее число в Option:

```
let next_optional_number = match optional_int {
    // В типе Option содержался None, присваиваем в next_optional_number
    // вариант None
    None => None,
    // В типе Option содержалось какое-то число. Увеличим его
    Some(number) => Some(number + 1)
}
```

Работает, но получается многословно. Есть типовая функция преобразования содержимого типов-контейнеров, в разных языках она может называться map(), transform() и т.д. С ней код получился бы гораздо лаконичнее:

```
let source_value = Some(42)

let next_value = source_value.map(|v| v + 1)
```

К подобным стандартным функциям можно отнести:

- map()
- filter() / grep()
- fold() / reduce()
- zip()

и пр. Функции раскрывают возможности паттерна программирования chaining methods:

```
let a = b
    // Когда определены b и c, посчитать b*2 + c
    .map(|v1| c.map(|v2| (v1 * 2) + v2))
    // Из Some(Some(result)) сделать Some(result). Из Some(None) или None сделать
None
    .flatten()
    // Извлечь из Option-результата значение, а иначе посчитать ...
    .unwrap_or_else(|| c.unwrap_or(24) * d.unwrap_or(42))
```

Краткое описание функций:

- .flatten() для Option-типа схлопывает конструкцию Some(Some(v)) в Some(v), а Some(None) и None — в None. Убирает вложенность.

- .unwrap_or() извлекает значение из Some(), а в случае None возвращает переданное значение.
- .unwrap_or_else() аналогично unwrap_or(), но в случае None возвращает результат работы переданной функции.

На Golang аналогичный алгоритм выглядел бы примерно так:

```
if b != nil {
    if c != nil {
        a = (*b * 2) + *c
    } else {
        if d != nil {
            a = 24 * *d
        } else {
            a = 24 * 42
        }
    }

} else {
    if d != nil {
        if c != nil {
            a = *c * *d
        } else {
            a = 24 * *d
        }
    } else {
        if c != nil {
            a = *c * 42
        } else {
            a = 24 * 42
        }
    }
}
```

Конструкцию можно сократить с помощью дополнительных присваиваний или написания вспомогательных функций, но на реальных данных это не всегда эффективно и возможно.

4.2 Явное лучше скрытого

Часто бывает так, что в популярную структуру надо добавить новое поле:

```
type UserAccount struct {
    // ...
    Balance int
}
```

Как убедиться, что во всех местах при создании объекта инициализировали новое поле, и пользователю не покажут нулевой баланс (значение поле по умолчанию)? Хорошо если у нас есть конструктор объекта, и структура создается всегда через него: достаточно поправить только конструктор. Но будем реалистами, качество кода не всегда идеально, и часто поля инициализируют кто как может.

Проблема решается одним движением: запрет на уровне языка на умолчательные значения полей. Иначе говоря, каждый раз, когда мы объявляем структуру, мы её обязаны объявить целиком.

4.3 Контроль изменений

Сидит замученный программист ядра. На его очень-важном сервере раз в несколько месяцев кто-то меняет значение поля в структуре, из-за чего контроллер iSCSI сходит с ума и большой NAS выходит из строя. Перед программистом миллионы строк кода, которые нельзя даже загнать в gdb. Как ему узнать, кто меняет поле?

Вероятно, подход должен быть противоположным: по умолчанию все данные должны стать read-only, и компилятор должен пресекать любые попытки их изменения. До тех пор, пока какие-то значения явным образом не объявили writable. Это решает множество архитектурных проблем (привет фунциональному программированию!), в том числе и с многопоточным программированием.

Здесь же можно сказать о передаче в функцию по значению. Многие языки позволяют менять содержимое переданного параметра — его локальной копии, после чего приходится рядом изобретать линтеры, которые предупреждают о возможной ошибке, которая не всегда является ошибкой.

4.4 Инфраструктура языка

4.4.1 Cargo

Со времен появления первых языков человечество изобретает инфраструктуру сборок. Самодеятельность со скриптами, Makefile, pkg-config, autoconf, autotools, CMake, npm, opam, cabal, ... В Rust для этого существует cargo. Но на самом деле, это больше, чем менеджер зависимостей и сборок. Некоторые из его возможностей:

- Стандартизованный механизм публикации библиотек на crates.io и docs.rs
- Вендоринг зависимостей
- Менеджмент зависимостей по стандарту Semantic Versioning
- Генерация документации
- Бенчмарки
- Юнит-тесты
- Автоматическое исправление кода: применение предложений линтера

- Генерация дерева зависимостей
- Шаблонизированное создание новых проектов

Кроме того, cargo поддерживает систему плагинов. Их сотни, приведу в пример некоторые из них:

- Статический анализатор
- Аудит зависимостей
- Отслеживание неиспользуемых и устаревших зависимостей
- Генератор новых приложений по шаблону
- Анализ покрытия кода тестами
- Слежение за изменениями кода и запуск каких-либо действий
- Упаковщики проекта в *.deb и *.rpm
- Генератор README.md

4.4.2 FFI и биндинги в C-библиотеки

Несмотря на разнообразие нативных библиотек для Rust, иногда нужно организовать взаимодействие и с библиотеками на языке Си. В Rust это автоматизировано через крейт bindgen, который выполняет кодогенерацию биндингов.

4.4.3 Тестирование

Необходимость в тестах в Rust довольно низкая, ввиду высокой степени доверия к компилятору. Тем не менее, создана органичная инфраструктура для разработки юнит-тестов, через дополнительные крейты поддерживающие в pytest-стиле и параметризацию и пр. Пример параметризованного теста:

```
#[rstest]
#[case(0, "0")]
#[case(1, "1")]
fn test(#[case] number: u32, #[case] expected: &str) {
    assert_eq!(expected, format!("{}", number))
}
```

Стоит обратить внимание и на проект cargo-nextest.