12 Новых языков программирования, которые вы должны знать

24.05.2023

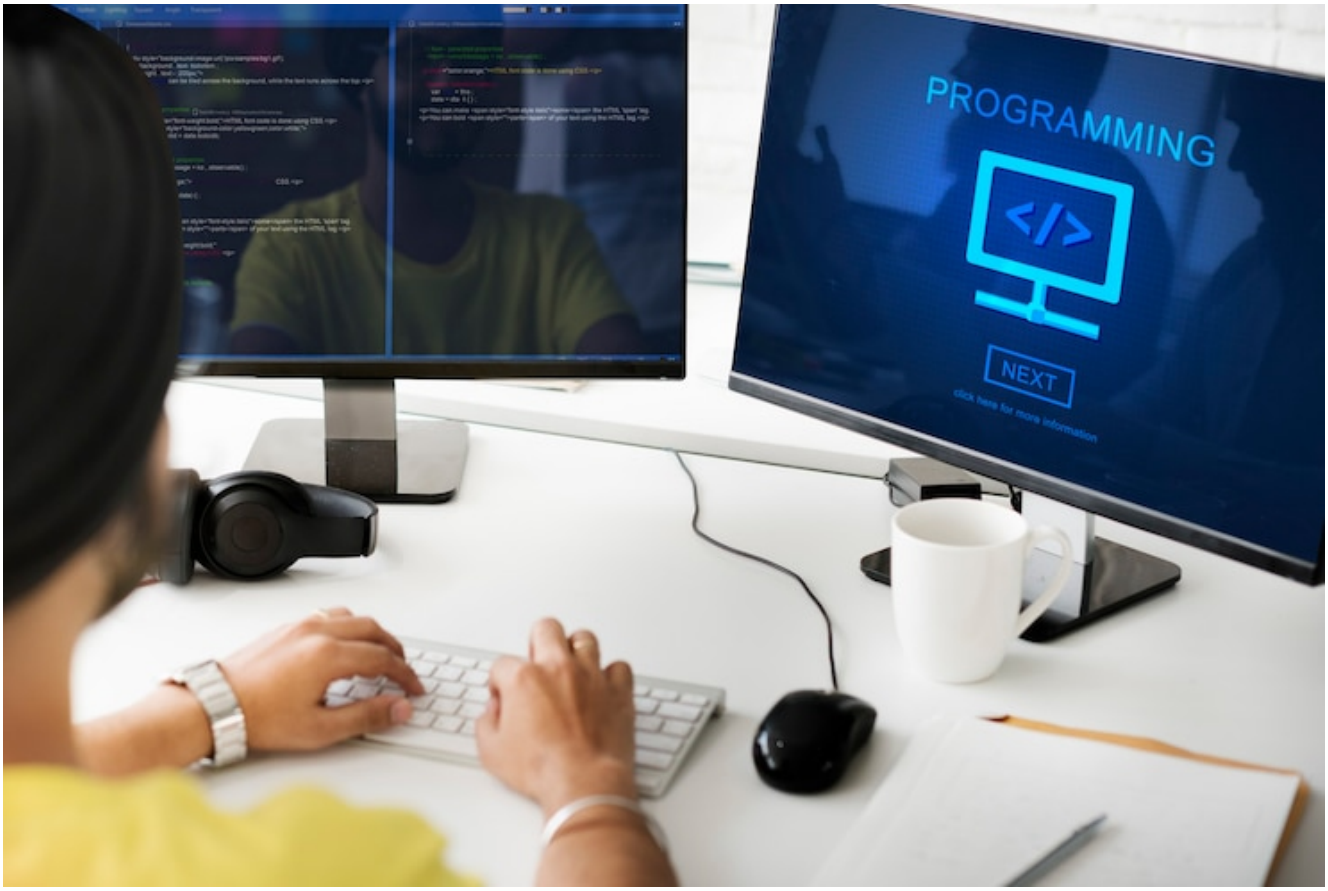
Для некоторых людей изучение нового языка программирования — это излишество, которого лучше избегать. Зачем, спрашивают они, нам нужен еще один язык? Разве у нас уже не достаточно языков? Разве существующие инструменты не справляются со своей работой? А еще есть неутомимые искатели, которые не упускают возможности попробовать что-то новое и наткнуться на драгоценные камни. Обучение — это самоцель, и их не нужно долго убеждать, чтобы потратить свои выходные на создание карточной игры для консоли на этом непонятном новом языке.

Независимо от того, к какому лагерю вы принадлежите, есть кое-что, не подлежащее обсуждению: мы все хотим тратить время на то, что будет иметь светлое будущее. Для тех, кто относится к

лагерю “наследия”, их любимый существующий язык уже набрал обороты и будет иметь светлое будущее. Для них я должен напомнить, что их уже ставший зрелым инструментарий разработки когда-то был “крутым напитком”, который многие отказывались пить. Так и сегодня; новые технологии добавляются, чтобы решить новые проблемы или облегчить боль при разработке.

Если что-то делает это достаточно радикально, есть шанс, что оно продолжит захватывать умы и достигнет вершины. Когда это произойдет, вы не захотите остаться позади. А для тех, кто любит бегать с блестящими штучками, предупреждение: веселье необходимо как программисту, но мы должны быть осторожны, чтобы не растрачивать нашу энергию бесцельно. Конечно, Brainfuck – это невероятно извращенный, головоломный, забавный маленький язык, но вы вряд ли получите пользу, занимаясь им серьезно. Вам нужно что-то новое, разумное и имеющее прочную основу.

Как были выбраны эти языки



Выбор языка – это пугающая работа, особенно когда новые языки

рассматриваются с точки зрения будущих преимуществ при трудоустройстве и удовлетворении потребностей. Каждый автор языка твердо уверен, что он создал идеальный язык и решил все проблемы, на которые только можно найти ответ. Как же тогда сократить? В этом посте я сосредоточился на нескольких параметрах, чтобы удержать свой поиск в разумных пределах.

Дата выпуска

Я специально избегал очень, очень новых языков. Под новыми я понимаю языки, которым на момент написания статьи менее 5-6 лет, и особенно те, которые еще не достигли стабильности (релиз 1.0, то есть). Но я надеюсь когда-нибудь вернуться к этому и написать об их успехах. Для меня 5-12 лет — это оптимальный срок, когда язык стабилизировался, и в него вносятся все новые и новые усовершенствования. Конечно, из этого правила есть исключения, и они будут рассмотрены в соответствующих случаях.

Сильный интерес сообщества

Этот пункт не вызывает сомнений, но часто игнорируется, когда мы радуемся чему-то новому. Для многих поддержка крупной компании является достаточным прецедентом для успеха, но это не всегда так. Да, Objective-C от Apple и теперь Swift процветают, потому что это были единственные варианты в строго контролируемой экосистеме, но D и Hack от Facebook (на мой взгляд, уродливый вариант и без того уродливого языка) остаются не более чем экспериментами.

Идеальной комбинацией был бы язык, поддерживаемый крупной, стабильной компанией и набирающий популярность (как React). Но суть вопроса все еще заключается в сообществе. Если язык не вызывает ажиотажа, и нет достаточного количества разработчиков для обучения и популяризации, он не вылезет из могилы на GitHub. Для меня это исключает такие зрелые, интересные языки, как Racket и Erlang, так как они остаются плоскими на кривой роста.

Сфокусированное, четко сформулированное USP

Третье, что я ищу, — это четко сформулированная цель языка. Например, улучшенная версия PHP, которая исправляет проблемы с именованием и компилируется в родной PHP, мне не подходит. Выгоды слишком малы, чтобы оправдать трещины, если все попытаются перейти на нее. По этому критерию я не могу не отвергнуть такие попытки, как CoffeeScript, и многие другие подобные языки, компилируемые в JavaScript.

Честно говоря, я сразу же отброшу улучшения, связанные только с синтаксисом. История показала, что полезность превосходит дружелюбие, когда речь идет о языках, поэтому если все, на чем фокусируется новый язык, — это более приятный опыт, боюсь, он не попадет в этот список.

Четкий, последовательный синтаксис

Да, я знаю. Я только что сказал, что синтаксис не имеет значения. Точнее, я сказал, что “улучшения только синтаксиса” не имеют значения. Но, несмотря на это, синтаксис остается одним из значимых факторов производительности программиста и удобства сопровождения, поэтому он обязательно будет фигурировать в моих расчетах.

Наряду с синтаксисом существует идея согласованности. Именование функций, структурирование модулей и т.д. — это ключевые вещи, которые язык не может позволить себе сделать неправильно. Сейчас 2018 год, ради бога, и еще один дизайн языка, похожий на PHP, был бы позором для всех нас. Итак, без лишних слов, давайте начнем с нашей пятерки лучших рекомендаций.

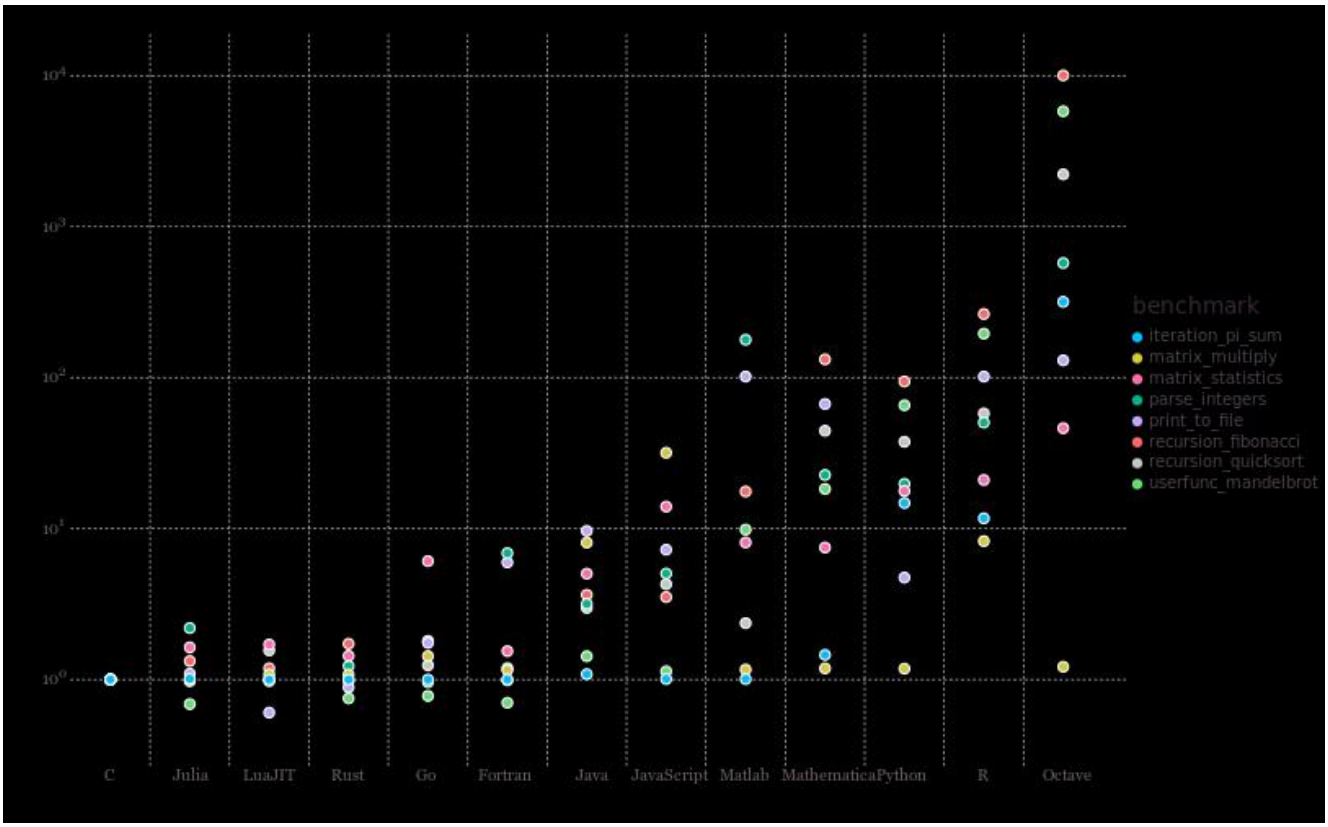
Julia

Для сотрудников Массачусетского технологического института языки, доступные для науки о данных, были в лучшем случае компромиссом. Python, при всей его ясности и популярности, не имел встроенных конструкций для работы с объектами данных. Кроме того, будучи интерпретируемым языком, Python был медленным для них (не библиотеки, заметьте, поскольку они в основном написаны на C). Но, возможно, самым большим недостатком была неуклюжая модель параллелизма и отсутствие модели параллельных вычислений, последняя из которых является основным элементом суперкомпьютеров. В результате родилась

Julia. Julia достигла своего стабильного релиза 29 сентября 2018 года, всего за несколько дней до написания этой статьи (поговорим о времени!). Вот что официальный сайт говорит о ее возможностях:

> *Julia превосходно справляется с численными вычислениями. Ее синтаксис отлично подходит для математики, поддерживается множество числовых типов данных, а параллелизм доступен из коробки. Множественная диспетчеризация в Julia является естественной для определения числовых и массивоподобных типов данных.*

С точки зрения синтаксиса, Julia можно рассматривать как комбинацию Python и C. Да, это, вероятно, первый случай, когда язык нацелился на оба этих качества и преуспел в этом. Чтобы упростить ситуацию, Julia работает молниеносно и не применяет строгую типизацию. Вот несколько эталонных примеров:



Как вы можете видеть, Julia немного хуже, чем C, но превосходит Java и Python. И что же такого интересного предлагает Julia?

Ну, вот реализация функции для вычисления гипотенузы правильного треугольника:

```
function hypot(x,y)
    x = abs(x)
    y = abs(y)
    if x > y
        r = y/x
        return x*sqrt(1+r*r)
    end
    if y == 0
        return zero(x)
    end
    r = x/y
    return y*sqrt(1+r*r)
end
```

Наконец, хотя большая часть экосистемы Julia в значительной степени ориентирована на математическую работу, я считаю, что у него большое будущее в области общего назначения. Насколько мне известно, это первый язык с первоклассной поддержкой параллельных вычислений, поэтому не будет ничего удивительного в том, что он приобретает все большую популярность в сфере Web и IoT.

Rust

Если вы пробовали новые версии браузера Firefox, вы знаете, что наконец-то, после нескольких лет работы, похоже, что они смогут отнять часть рынка у Chrome. Если браузер кажется легким, шустрым и быстро рендерится, это все благодаря языку, специально разработанному Mozilla: Rust. Сказать, что у Rust большое будущее, будет ложью; язык уже пользуется огромным успехом, и если вы еще не слышали о нем, то это потому, что область его применения специализирована, а его цель пугает: заменить C++! Да, наконец-то у нас есть язык, который не только способен на это, но и уже делает это. Для людей, разочарованных перегруженным дизайном C++ и проблемами управления памятью, Rust — это глоток свежего воздуха.

Вот как выглядит программа на языке Rust:

```rust
fn is_odd(n: u32) -> bool {
    n % 2 == 1
}

fn main() {
    println!("Find the sum of all the squared odd numbers under 1000");
    let upper = 1000;

    let sum_of_squared_odd_numbers: u32 =
        (0..).map(|n| n * n)                             // All natural numbers squared
             .take_while(|&n_squared| n_squared < upper) // Below upper limit
             .filter(|&n_squared| is_odd(n_squared))     // That are odd
             .fold(0, |acc, n_squared| acc + n_squared); // Sum them
}
```

На мой взгляд, лаконично и элегантно. Rust следует подходу функционального программирования, что делает ваш код более композиционным, и в нем нет объектно-ориентированных иерархий, с которыми нужно бороться. Итак, что же дает Rust смелость идти за C++? Это новая модель памяти. Вместо того чтобы полагаться на старый танец new()/delete(), Rust вводит идею владения. Вместо того чтобы выделять память и обращаться к ней напрямую, переменные в Rust “заимствуют” друг у друга, причем компилятор накладывает на них строгие ограничения. Общая концепция слишком сложна, чтобы объяснить ее в нескольких словах, поэтому не стесняйтесь заглянуть в официальную документацию, чтобы узнать больше. Суть в том, что это приводит к 100% сохранности памяти без необходимости использования сборщика мусора, что очень важно.

Rust захватил мир системного программирования. Он уже поддерживается на некоторых платформах, браузеры и движки рендеринга быстро заменяют код на C/C++ в производственных системах, его используют для написания операционных систем. Конечно, не всем по душе создавать еще один браузер или драйвер устройства, но Rust уже распространяется в других областях. У нас уже есть несколько полнофункциональных и до смешного быстрых веб-фреймворков на Rust, и разрабатывается все больше и больше библиотек приложений. Честно говоря, если вас интересует захватывающее будущее, Rust — идеальный язык, и сейчас для этого самое подходящее время. Rust — это самолет, который уже взлетел, но еще есть время подняться на борт, пока он летит к звездам!

Elixir

Среди языков, ориентированных на счастье разработчика, первое место навсегда закрепилось за Ruby. Это язык, который читается как поэзия и имеет достаточно ярлыков, чтобы уменьшить умственное трение на порядок. Неудивительно, что фреймворк Rails продолжает доминировать в разработке полного стека для серьезных разработчиков и стартапов. Но не все были довольны Rails, особенно один из его основных разработчиков — Жозе Валим. Я думаю, что сам создатель лучше всего объясняет генезис этого языка в одном из интервью:

> *Это длинная история, но я постараюсь изложить ее коротко и ясно. В 2010 году я работал над улучшением производительности Rails при работе с многоядерными системами, поскольку наши машины и производственные системы поставлялись со все большим количеством ядер. Однако весь этот опыт был довольно разочаровывающим, поскольку Ruby не предоставляет надлежащего инструмента для решения проблем параллелизма. Тогда я начал присматриваться к другим технологиям и в конце концов влюбился в виртуальную машину Erlang. Я начал использовать Erlang все больше и больше и с опытом заметил, что мне не хватает некоторых конструкций, доступных во многих других языках, включая функциональные. Именно тогда я решил создать Elixir, как попытку привнести различные конструкции и отличный инструментарий поверх виртуальной машины Erlang.*

И вот, Elixir родился! Подобно тому, как Scala улучшает язык Java, но использует ту же виртуальную машину (JVM), Elixir использует преимущества проверенной десятилетиями виртуальной машины Erlang. Обсуждение Erlang выходит за рамки данной статьи, но минимум, что вы должны знать, это то, что это самый сокровенный секрет телекоммуникационной индустрии: если наши телефонные сети намного надежнее наших веб-систем, то это все благодаря Erlang.

Если говорить еще проще, то это означает следующее. Если вы

создаете систему реального времени, например, чат, Elixir гораздо менее требователен к оперативной памяти и стабилен, чем Ruby (или PHP, Python и Java, если уж на то пошло). Машина, на которой работает Ruby и максимальное число одновременных подключений составляет 10 000, может легко выдержать 200 000 при использовании Elixir и при этом иметь достаточно оперативной памяти для запуска 2D-игр!

```
30  def start_link(name, opts \\ []) do
31    GenServer.start_link(__MODULE__, {name}, opts)
32  end
33
34  def init({name}) do
35    require Logger
36    Logger.log :debug, "Started channel #{name}!"
37    :pg2.join(:channels, self)
38    :ets.insert(:channels, {name, self})
39    users = :ets.new(:users, [:set, :protected])
40    {:ok, {name, users, []}}
41  end
42
43  def handle_call({:send, message}, _from, {name, _users, _buffer} = state) do
44    Kaguya.Util.sendPM(name, message)
45    {:reply, :ok, state}
46  end
47
48  def handle_call({:rename_user, {old_nick, new_nick}}, _from, {_name, users, _buffer} = state) do
49    case :ets.lookup(users, old_nick) do
50      [{^old_nick, user}] ->
51        new_user = %{user | nick: new_nick}
52        :ets.delete(users, old_nick)
53        :ets.insert(users, {new_nick, new_user})
54      [] -> :ok
55    end
56    {:reply, :ok, state}
57  end
```

С точки зрения синтаксиса Elixir беззастенчиво копирует Ruby, а его доминирующий веб-фреймворк Phoenix беззастенчиво копирует Rails. Я бы сказал, что это и хорошо, потому что вместе с Laravel, Grails, Masonite и т.д. мы достигаем того момента, когда все языки имеют Rails-подобные фреймворки, которые могут облегчить переход. Кто-то может посмеяться над “отсутствием оригинальности”, но я, по крайней мере, не жалуюсь. Наконец, Elixir — одна из тех технологий, которые освежают, радуют и чертовски практичны. Несколько Ruby (и даже не Ruby) магазинов переходят на Elixir, а крупные компании, такие как Pinterest, используют его в производстве с

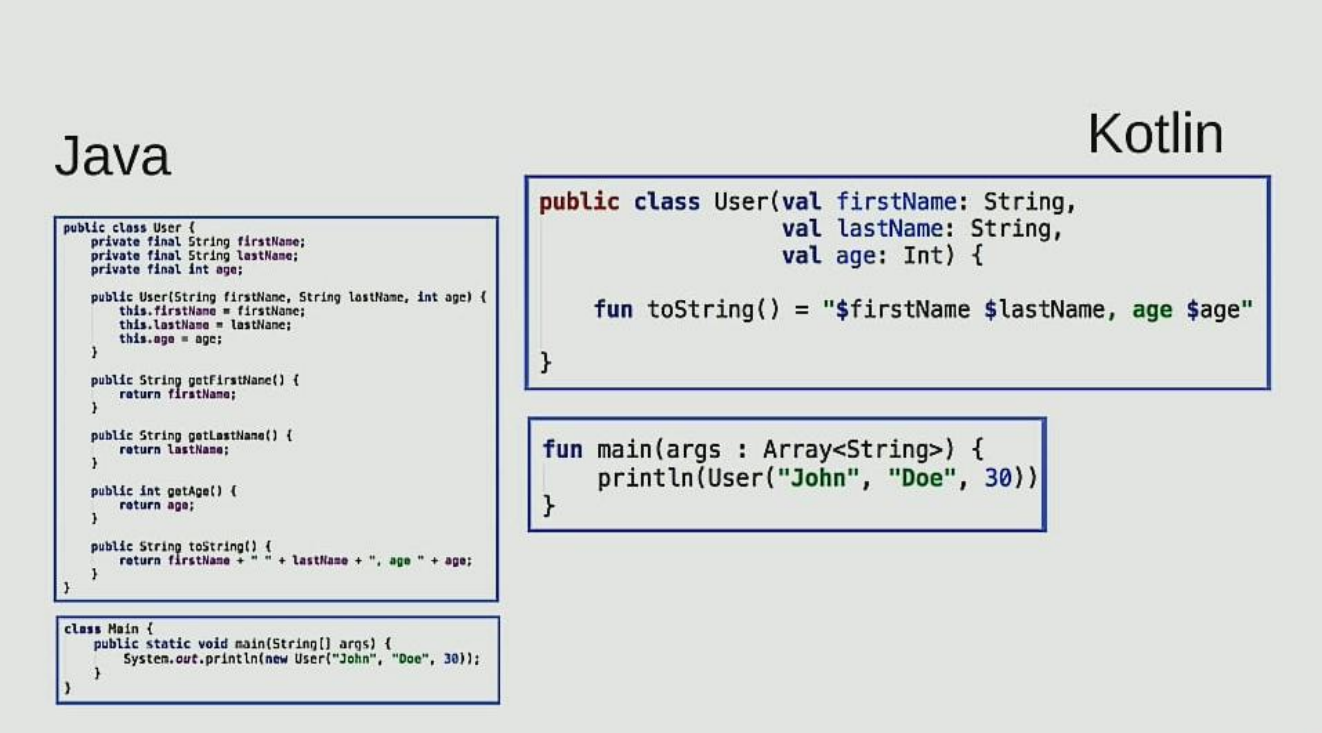
чрезвычайно удовлетворительными результатами. Многие считают, что Node.js был импровизированной попыткой параллелизма и скоро будет заменен Elixir. Должен сказать, что я с ними согласен.

Kotlin

В 2017 году на I/O компания Google выпустила бомбу на ничего не подозревающую толпу. Компания официально объявила Kotlin в качестве основного языка для разработки Android, что вызвало шок в индустрии. То, что Google активно ищет замену Java, неудивительно после того, как компания была укушена иском Oracle; однако принятие Kotlin было несколько неожиданным, и все еще существует большая вероятность того, что Google вскоре выпустит свою виртуальную машину. Тем не менее, на данный момент Kotlin переживает бурный рост.

Kotlin был разработан компанией JetBrains, более известной своим набором безумно хороших редакторов кода. Один из них, IntelliJ IDEA, является основой Android Studio. Целями разработки Kotlin являются безопасность, лаконичность и 100% совместимость с Java. Более всего компилятор Kotlin старается исключить исключения нулевого указателя, которые так часто встречаются в мире Java. Он также значительно снижает пресловутую многословность Java, что для многих станет облегчением.

Вот замечательное сравнение кода на Java и Kotlin:



Код Kotlin значительно короче, и в нем гораздо меньше когнитивных перегрузок. Но давайте проясним одну вещь: Kotlin вряд ли заменит Java, хотя он быстро становится фаворитом. Я считаю, что через десять лет малые и средние команды будут смотреть в сторону Kotlin, в то время как большие группы будут продолжать использовать Java исключительно по причинам наследия. Тем не менее, у Kotlin очень большое будущее, поскольку он делает все то же самое, что и Java, может сливаться с кодом Java незаметно для всех и гораздо более приятен!

TypeScript

Видит Бог, мне пришлось сдерживать себя в этом месте! Все внутри меня кричало: “Elm! Elm!”, но независимо от того, насколько революционны его идеи или насколько божественен синтаксис, Elm пока не рассматривается в качестве основной альтернативы для front-end работы. В любом случае, давайте перейдем к тому, что является основной: TypeScript. JavaScript — это как дикие ягоды: уродливые и неприятные, но вы должны их есть, если хотите выжить в джунглях фронтенд-разработки. Было предпринято много попыток заменить его (и, скорее всего, новый стандарт WebAssembly увенчается успехом), но то, что

действительно привлекло всеобщее внимание, — это супер-множество, разработанное Microsoft.

Велика вероятность, что вы слышали о TypeScript: Angular был первым фреймворком, который использовал его начиная со второй версии, и люди быстро приняли это к сведению. Это потому, что TypeScript добавляет некоторые столь необходимые и фантастические супер-способности в самый известный язык программирования в мире. Да, наконец-то стало возможным писать родной код на JavaScript, не мучаясь и не проклиная свое рождение!

Вот какие усовершенствования привносит TypeScript:

- Сильная типизация: Наконец-то строка — это не число, а число — это не объект, который не является пустым массивом!
- Проверка типов во время компиляции: Если ваш код компилируется правильно, то он более или менее гарантированно свободен от недостатков среды выполнения JavaScript.
- Классы и модули: Да, классы являются стандартом в ES6, но они также включены в TypeScript, помимо аккуратной системы модулей.
- Вывод типов: Для сложных типов тип может быть легко определен компилятором, что избавит вас от головной боли.
- Async/await: ключевые слова и паттерны async/await являются фундаментальными, так что больше не нужно возиться с Promises и Callbacks!

Пространства имен, дженерики, кортежи… . . Я мог бы продолжать и продолжать, но достаточно сказать, что TypeScript превращает один из худших опытов разработки в один из лучших.

```
class Animal {
    private name: string;
    constructor(theName: string) { this.name = theName; }
}

class Rhino extends Animal {
    constructor() { super("Rhino"); }
}

class Employee {
    private name: string;
    constructor(theName: string) { this.name = theName; }
}

let animal = new Animal("Goat");
let rhino = new Rhino();
let employee = new Employee("Bob");

animal = rhino;
animal = employee; // Error: 'Animal' and 'Employee' are not compatible
```

Влияние TypeScript невозможно отрицать. Он вытеснил с поля такие попытки, как Dart от Google (хотя он пытается вернуться через Flutter, фреймворк для мобильной разработки), и открыл глаза разработчикам JS на преимущества более сильных типов. В результате такие важные библиотеки, как React, D3, Vue (даже jQuery!), теперь имеют версию TypeScript, а в лучших магазинах программного обеспечения по всему миру весь код JavaScript пишется как код TypeScript. Заголовки TypeScript теперь доступны и для Node.js (честно говоря, если нода сможет улучшить свою историю с параллелизмом и исправить свое паршивое управление памятью, она будет существовать вечно).

Возможно, вы удивитесь, узнав, что создатель Node.js, публично пожалев о своем творении, работает над новой средой исполнения (сейчас нет официального сайта, только репозиторий на GitHub), основным языком которой является TypeScript. Лучшие новости? TypeScript — это небольшой язык для изучения с существенными преимуществами в будущем. Если вы являетесь разработчиком JavaScript среднего уровня, то за два дня вы освоите достаточно TypeScript, чтобы перенести весь существующий код!

Elm

Elm, как быстро произносится его название, используется для создания фронтенд-приложений, графики и игр. Это функциональный язык программирования, созданный Эваном Чаплицки в 2012 году. Считается, что у Elm нет исключений во время выполнения – вот где он сияет. Как статически типизированный язык, компилятор проверяет все ошибки во время компиляции (ничего себе!) с дружественными сообщениями (сообщения представляют собой полные тексты, а не странные брошенные коды).

Какой вздох облегчения для разработчиков (дебаггеров)! Вы сможете научиться кодить на Elm, так как будете получать все больше ошибок – компилятор скажет вам, что не так, и предложит, что нужно сделать, чтобы исправить это! Elm может похвастаться тем, что он быстрее, чем React, и более сильно типизирован, чем даже TypeScript. Код Elm очень организован и аккуратен и сделает вас лучшим разработчиком.

Поскольку Elm основан на модулях, вы можете легко создавать многократно используемые компоненты. Elm компилируется в JavaScript, который можно запускать в браузере. Итак, все, что вам нужно для работы Elm, это node и npm, и вы можете просто получить Elm с помощью команды:

```
npm install -g elm@<version>
```

Вы можете поставить версию, которую хотите установить, например, 0.19.1. Затем вы можете проверить, установлен ли Elm должным образом, используя команду ***-version***. Если вы не хотите пока заниматься установкой и настройкой – просто зайдите на их официальный сайт и воспользуйтесь онлайн-редактором, чтобы поиграть. Итак, давайте немного поиграем! Если вы не используете онлайн-компилятор, вам придется установить все зависимости для программы, которую мы сейчас напишем (впрочем, это довольно просто).

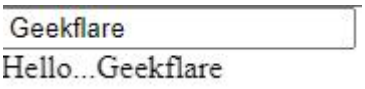
Попросим пользователя ввести свое имя в текстовое поле и вывести его на страницу с приветствием.

```
import Browser

import Html exposing (Html, Attribute, div, input, text)

import Html.Attributes exposing (..)

import Html.Events exposing (onInput)

-- MAIN

main =

  Browser.sandbox { init = init, update = update, view = view
}

-- MODEL

type alias Model =

  { content : String

  }

init : Model

init =

  { content = "" }

-- UPDATE

type Msg

  = Change String

update : Msg -> Model -> Model

update msg model =
```

```
  case msg of

    Change newContent ->

      { model | content = String.append "Hello..." newContent
}

-- VIEW

view : Model -> Html Msg

view model =

  div []

    [ input [ placeholder "Type your name", onInput Change ]
[]

    , div [] [ text (model.content) ]

    ]
```

Вот начальный экран, когда вы создаете программу:

Введите имя, и вот что вы получите на экране:



Хотя эта программа может показаться чрезмерно сложной для своей цели, по мере увеличения сложности программы вы оцените, насколько легко ее отлаживать и поддерживать. Вы можете увидеть четкое разделение между моделью, представлением и контроллером (обновлением). Подобно тому, как мы используем теги HTML, мы можем создавать формы в Elm с помощью тегов model div. По событию ‘on input’ (т.е. пользователь вводит текст) программа вызывает ‘Change’, и программа печатает имя пользователя вместе с ‘Hello’ с помощью функции String.append.

Pony

Pony скомпилирован и следует акторной модели вычислений, разработанной для асинхронного поведения – т.е. для высоко параллельных приложений. Традиционные языки программирования предоставляют функцию “блокировки” для обработки параллелизма, что влияет на производительность. В Pony нет блокировки, что позволяет избежать блокирующих операций или тупиковых сценариев. Каждый агент является однопоточным.

Pony также обеспечивает безопасность на основе возможностей, где пользователи должны использовать “ссылочные возможности” для доступа к определенному объекту, что обеспечивает безопасную работу с данными. Например, возможности описывают то, что другим псевдонимам запрещено, а не то, что им разрешено. Такие понятия, как изменяемость и изоляция, основаны на этих возможностях. Эта функция “запрета” возможностей делает Pony свободным от гонки данных. Pony безопасен, быстр и точен и экономит время разработки, что делает его хорошим выбором для банковских и финансовых приложений.

Pony обеспечивает безопасность типов при работе с данными. В нем нет исключений – для компиляции кода необходимо обрабатывать только ошибки. Основной причиной этого является то, что Pony статически типизирован. Вам необходимо явно указать тип (как в Java и в отличие от Python) переменной перед ее использованием:

```
let name: String
```

Как и в Java, вы можете создавать конструкторы. Предположим, у вас есть класс служащего с именем и возрастом:

```
class Employee

  let name: String

  let age: U64
```

```
  new create(name': String) =>

    name = name'
```

Разработчики Java могут заметить, что в синтаксисе есть тонкие различия (нет фигурных скобок, вот это да!). В конце атрибутов класса также есть символ ”’. Вы также можете создавать функции:

```
fun get_emp_name(): String => name
```

Весело писать функции, да? Теперь о главном – об акторах и обещаниях. У Pony actors есть Behaviours – похожие на функции, но только асинхронные – они выполняются в какое-то время в ближайшем будущем, но не обязательно сразу при вызове. Но они “обещают”, что поведение будет выполнено обязательно.

```
actor Employee
  // actor has fields, similar to class
  let name: String

  // and of course, constructor
  new create(name': String) =>
    name = name'

  // Note the behaviour 'be' instead of the fun function
  be get_emp_name(promise: Promise[String]) => promise(name)
```

Обещания тоже могут быть отклонены – если актор не может выполнить отправленный асинхронный запрос. Вы можете создать обещание:

```
// Create a new promise

let promise = Promise[String]
```

И вызовите поведение актора, передав обещание, которое должно быть выполнено (в нашем случае, получение имени сотрудника).

```
employee.emp_get_name(promise)
```

Как вы думаете, скомпилируется ли этот код? Еще одна замечательная особенность Pony заключается в том, что он безопасен для памяти – никаких NULL или переполнений буфера. Любой код, который может вернуть null, никогда не будет компилироваться, пока не будет исправлен.

Ну, мы еще не сказали программе, что такое ’employee’ (наш агент):

```
let employee = Employee(“J K Rowling”)
```

В отличие от любого другого языка программирования, Pony позволяет делить на ноль, и результат равен нулю. Для всех возможностей языка Pony существуют математические доказательства. Чтобы писать программы на языке Pony, необходимо установить компилятор Pony. Поскольку Pony – компилируемый язык, перед запуском программы вы должны скомпилировать ее с помощью ‘ponyc’. Как только вы установите компилятор, попробуйте написать программу hello world.

Vyper

Vyper – это язык программирования смарт-контрактов, основанный на Python. Как и Python, он является человеко-читаемым, простым в написании и безопасным. Vyper компилируется до байткода виртуальной машины Ethereum (EVM). EVM определяет состояние Ethereum для каждого блока в блокчейне. Чтобы понять, почему Vyper особенный, давайте разберемся, что такое смарт-контракты. Смарт-контракты – это программы, хранящиеся в блокчейне, которые определяют и исполняют соглашение между продавцом и покупателем при выполнении требований контракта.

Это само-исполняющиеся автоматизированные контракты, которые не нуждаются во вмешательстве человека. Однако смарт-контракты подвержены уязвимостям. Например, смарт-контракты можно заставить выпустить эфир на произвольные адреса или убить произвольные адреса, или они могут быть не в состоянии выпустить эфир. Эти уязвимости обычно появляются в коде –

непреднамеренно или намеренно. Vyper устраняет эту проблему, предоставляя безопасный код, что затрудняет внедрение уязвимого или вводящего в заблуждение кода. Хотя Vyper основан на Python, он не следует многим парадигмам ООП, таким как наследование, перегрузка, рекурсия и т.д. Это позволяет избежать усложнения кода (наличие нескольких файлов, что затрудняет аудит).

Vyper также не поддерживает встроенный ассемблер, что означает, что программы не могут выполнять какие-либо действия непосредственно на EVM, что позволяет избежать атак. Эти особенности делают Vyper вполне безопасным для написания кода смарт-контрактов, используемых в блокчейне. Чтобы попрактиковаться в написании программ на Vyper, вы можете воспользоваться онлайн-компилятором remix. Вы также можете установить Vyper с помощью docker или pip (если у вас есть Python), следуя инструкциям на странице документации Vyper.

R

R – один из самых популярных языков программирования для анализа данных и машинного обучения. Он имеет API для всех сложных математических, статистических и научных расчетов, алгоритмов машинного обучения и визуальных представлений. R имеет открытый исходный код и широко популярен благодаря своему богатому графическому пользовательскому интерфейсу. Он имеет активное и стабильное сообщество и легко интегрируется с другими языками, такими как C, C++ и т.д.

Все вышеперечисленные возможности мы получаем благодаря CRAN (Comprehensive R Archive Network), которая содержит более 10000 пакетов для статистики, вероятности, анализа данных, вычислений, графики и многого другого. Чтобы увидеть магию R, давайте попробуем выполнить простую программу для нахождения среднего (mean) из 11 чисел. Чтобы найти среднее значение, мы берем сумму чисел и делим ее на общее количество значений (в нашем случае 11). В R есть функция ‘mean’, которая выполняет

все эти вычисления за нас.

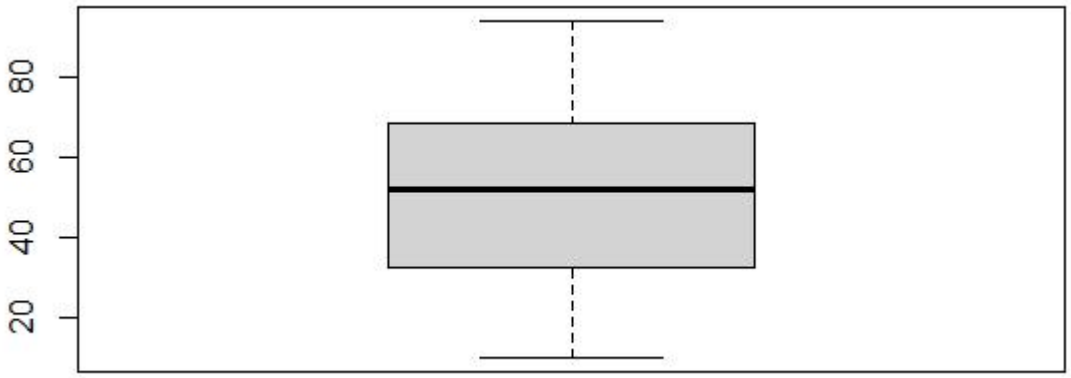
```
mynums <- c(51, 52, 53, 94, 88, 61, 31, 34, 76, 20, 10)

mean(mynums)
```

Вывод:

```
[1] 51.81818
```

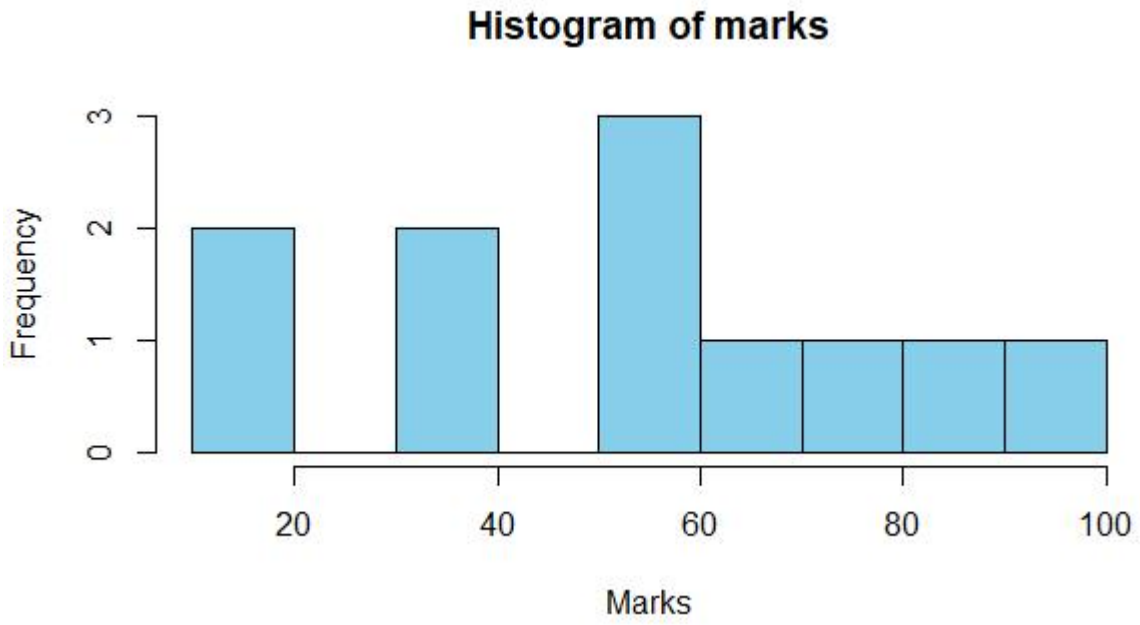
Мы можем построить их с помощью метода plot:

```
boxplot(mynums)
```



Существует множество расширенных пакетов, таких как ggplot2, dplyr и многие другие, для отображения насыщенных графиков в R. Мы также можем просмотреть быструю гистограмму с приведенными выше значениями, чтобы увидеть диапазон, в который попадают значения.

```
hist(mynums, breaks = 10, col = "sky blue", main = "Histogram of marks", xlab = "Height Bin")
```



Обратите внимание, что мы задали разбиение на 10; мы можем изменить его на любое число в зависимости от желаемых делений. Приведенная выше переменная mynum была вектором, содержащим список чисел. Как и в Python, в R мы создаем фрейм данных, чтобы работать с большим количеством измерений. Это наиболее полезно для аналитики. Например, мы можем объединить несколько векторов, создать фрейм данных и манипулировать им в зависимости от количества переменных и типа анализа, который нам нужен.

Допустим, у нас есть векторы имени сотрудника, квалификации и возраста. Мы можем создать фрейм данных и отобразить все данные вместе:

```
employees = data.frame(Name = c("John","Mac","April","Ron","Matt"),

Age = c(23,28,30,43,31), Skill = c("Java", "Python", "C++", "R", "PHP"))

print(employees)
```

```
> print(employees)

    Name Age  Skill
```

```
1   John  23   Java

2    Mac  28 Python

3  April  30    C++

4 Ron  43      R

5 Matt  31    PHP
```

Еще одна интересная особенность R – простота работы с матрицами с помощью массивов. R поразит вас, выполнив сложные матричные вычисления с легкостью. Все, что вам нужно сделать, – это создать матрицу и передать ее программе R.

```
M1 <- matrix(c(1, 2, 1, 2), ncol=2)

M2 <- matrix(c(3, 4, 3, 4), ncol=2)

print(M1*M2)

>print(M1*M2)

     [,1] [,2]

[1,]    3    3

[2,]    8    8
```

Apache Groovy

После тщательного рассмотрения и оценки я включил Groovy в свой список лучших языков программирования. Этот язык – как масло на вершине вкусного торта, придающее дополнительный вкус и улучшающее любой проект. Одна из главных причин, по которой Groovy заслуживает места в списке, – это его широкий спектр возможностей, которые способствовали росту его популярности в мире технологий. Будучи гибким и динамичным языком для виртуальной машины Java (JVM), он обеспечивает современные возможности программирования для Java-разработчиков с

минимальной кривой обучения.

Стоит отметить, что JVM — это аббревиатура от Java Virtual Machine. JVM является неотъемлемой частью Java, обеспечивая платформу для выполнения байткода Java на любом устройстве. Groovy, построенная на базе JVM, предлагает широкий спектр возможностей, которые повышают ее производительность и делают ее привлекательной для разработчиков. Проще говоря, он предоставляет платформу для выполнения байткода Java, что делает возможным использование Java на любом устройстве. С точки зрения производительности, Groovy может статически проверять типы и компилировать ваш код для повышения надежности и производительности. Легкая интеграция Groovy с существующими классами и библиотеками Java отличает его от других языков программирования.

Он также может компилироваться прямо в байткод Java, что делает его удобным для использования везде, где есть Java. Эта особенность повышает гибкость и универсальность Groovy и делает его лучшим выбором для разработчиков, желающих создать эффективное и надежное программное обеспечение. Динамическая природа Groovy может сделать проверку типов проблематичной, а отладку кода более сложной, что является одним из потенциальных недостатков. Однако многие программисты считают,

что гибкость и простота Groovy перевешивают все потенциальные недостатки. Как разработчик, вы должны признать уникальную ценность Groovy. Чтобы проиллюстрировать эту мысль, я хотел бы поделиться примером, который показывает, как мощные возможности Groovy могут улучшить ваш код.

```
def greeting = "Hello, World!"
println greeting

// Define a list of numbers
def numbers = [1, 2, 3, 4, 5]

// Use a closure to map the list to its squares
def squares = numbers.collect { it * it }

println squares
```

В этом примере мы иллюстрируем универсальность и выразительность языка Groovy, определяя строковую переменную и выводя ее на консоль с помощью функции println. Кроме того, мы показываем, как Groovy упрощает сложные операции, например, преобразование списка чисел с помощью замыкания для создания нового списка, содержащего квадраты каждого числа. Это демонстрирует, как Groovy может повысить вашу производительность как разработчика, предоставляя краткий, читабельный синтаксис для повседневных задач программирования. Таким образом, становится очевидным, почему Groovy заслуживает внимания каждого разработчика.

Crystal

После обширного исследования мы не могли не добавить Crystal в наш список. И нет, мы не говорим о минералах! Crystal – это объектно-ориентированный язык программирования общего назначения, выпущенный в 2014 году. Он был разработан, чтобы иметь синтаксис, близкий к Ruby, и при этом быть быстрым и эффективным. Благодаря статической системе типов и опережающей компиляции, Crystal предлагает разработчикам скорость C и

простоту Ruby. Crystal – относительно новый язык программирования, набирающий популярность среди разработчиков благодаря своей впечатляющей скорости и простоте использования. Его часто описывают как “Быстрый как C, ловкий как Ruby”, подчеркивая его способность обеспечивать молниеносную производительность при сохранении дружественного синтаксиса и читабельности Ruby.

Однако Crystal достигает своей впечатляющей скорости, жертвуя некоторыми динамическими аспектами Ruby и ограничивая некоторые программные конструкции. Тем не менее, этот компромисс сделал Crystal привлекательным вариантом для создания высокопроизводительных приложений на более удобном для разработчиков языке.

```
# Define a class for a person with name and age attributes
class Person
  getter name : String
  getter age : Int32

  def initialize(@name : String, @age : Int32)
  end
end

# Create an array of Person objects
people = [Person.new("Alice", 25), Person.new("Bob", 30),
Person.new("Charlie", 35)]
```

```
# Use a block to filter the array by age and map the names to uppercase
names = people.select { |person| person.age >= 30 }.map { |person| person.name.upcase }

# Print the resulting array of uppercase names
puts names.inspect
```

Этот код демонстрирует преимущества Crystal в синтаксисе, производительности и безопасности типов. Синтаксис Crystal похож на синтаксис Ruby, что делает его легким для чтения и написания. Однако Crystal компилируется в нативный код, что приводит к более быстрому выполнению, чем интерпретируемые языки, такие как Ruby. Кроме того, Crystal статически типизирован, что обеспечивает безопасность типов во время компиляции и повышает производительность. В этом примере код использует блок для фильтрации массива объектов Person по возрасту и преобразования имен в верхний регистр, демонстрируя гибкость и выразительность синтаксиса Crystal.

Reason

Reason — это современный язык программирования с синтаксисом, похожим на JavaScript или другие языки семейства C, и надежной системой типов OCaml. Благодаря сильным возможностям проверки типов, разработчики могут находить проблемы раньше и писать более надежный и безопасный код. Удобный интерфейс и простой дизайн Reason делают его фантастическим выбором для различных видов программирования, независимо от уровня вашего опыта программирования.

Reason и OCaml — очень универсальные языки программирования, которые можно использовать в различных средах благодаря многочисленным проектам, поддерживающим их. Одним из способов их использования является создание нативных исполняемых файлов, которые могут быть запущены непосредственно на вашей машине с помощью стандартного компилятора. Кроме того, существует несколько инструментов, таких как “dune” и “esy”, которые помогают в этом процессе. Другой вариант — компиляция Reason в JavaScript-код, совместимый с браузерами, что можно сделать с помощью таких проектов, как ReScript (бывший BuckleScript) и Js_of_ocaml. Эти универсальные возможности делают Reason и OCaml привлекательными для разработчиков в различных отраслях. Reason — это современный язык программирования с удобным интерфейсом и синтаксисом, похожим на JavaScript. Его универсальность и простой дизайн делают его популярным среди разработчиков в различных отраслях.

Заключение и отказ от ответственности

У каждого из нас есть свои предпочтения, но перечисленные выше языки стоит попробовать, поскольку они обладают всеми новыми возможностями и решают многие проблемы, оставленные предыдущими языками. Elm отлично подходит для более

аккуратного разделения кода и модульного программирования. Pony – один из лучших для асинхронного программирования. Хотя Vyper является довольно новым в мире языков смарт-контрактов, он представляет собой многообещающую альтернативу Solidity. Vyper особенно хорош, когда речь идет об определении контрактов и обработке ошибок. R, с другой стороны, считается одним из лучших языков для аналитики и уже имеет большое сообщество.

Отдельные языки набирают такую же популярность, как и те, о которых идет речь в этой статье, но по разным причинам не были включены в список. Вот краткий обзор:

- Golang: Уже зарекомендовал себя как основной, хотя и не очень популярный язык. Я считаю, что на данный момент у Golang есть несколько конкурентов, которые будут поддерживать его долю рынка на низком уровне.
- Swift: Apple железной хваткой держит свою экосистему, и Swift – единственный язык, доступный там. Раньше был в моде Objective C, как сейчас Swift. Я считаю это жульничеством и поэтому отказываюсь включать его сюда.

Будущее всегда неопределенно, и один из способов сделать карьеру – придерживаться того, что уже работает, и отказываться “отвлекаться”. Если вы так поступаете, то Java, PHP, Python, Ruby и т.д. – все это отличные языки, на которых можно остановиться. Однако для некоторых из нас нормы недостаточно. Они хотят исследовать и делать большие ставки на будущее. Если вы относитесь к последнему лагерю, то один из этих пяти языков должен быть на вершине вашего списка дел. Наконец, пытаясь оценить язык, не позволяйте усилиям захлестнуть вас, потому что их не так уж много. Если вы уже знаете несколько языков программирования, вы сможете выучить любой из них максимум за два месяца, уделяя 5-6 часов в неделю. В то время как счастье и денежная отдача, которые можно получить в будущем, будут в несколько раз больше.