Современные языки программирования

Содержание

Введение

Прогресс компьютерных технологий определил процесс появления новых разнообразных знаковых систем для записи алгоритмов – языков программирования. Смысл появления такого языка – оснащенный набор вычислительных формул дополнительной информации, превращает данный набор в алгоритм. Язык программирования служит двум связанным между собой целям: он дает программисту аппарат для задания действий, которые должны быть выполнены, и формирует концепции, которыми пользуется программист, размышляя о том, что делать.

Первой цели идеально отвечает язык, который настолько "близок к машине", что всеми основными машинными аспектами можно легко и просто оперировать достаточно очевидным для программиста образом. Второй цели идеально отвечает язык, который настолько "близок к решаемой задаче", чтобы концепции ее решения можно было выражать прямо и коротко. Связь между языком, на котором мы думаем/программируем, и задачами и решениями, которые мы можем представлять в своем воображении, очень близка. По этой причине ограничивать свойства языка только целями исключения ошибок программиста в лучшем случае опасно. Как и в случае с естественными языками, есть огромная польза быть, по крайней мере, двуязычным. Язык предоставляет программисту набор концептуальных инструментов, если они не отвечают задаче, то их просто игнорируют.

Выбранная тема является актуальной, так как системы программирования – это универсальные средства работы с информацией. С их помощью можно решать вычислительные задачи, обрабатывать тексты, получать графические изображения, осуществлять хранение и поиск данных и т.д., в общем, делать все, что делают средства прикладного программного обеспечения – специализированные исполнители. Кроме того, сами эти средства (графические и текстовые редакторы, СУБД и др.) – это программы,

написанные на языках программирования, созданные с помощью систем программирования.

Языки программирования претерпели большие изменения с тех пор, как в сороковых годах началось их использование. Они все еще продолжают изменяться и теперь даже быстрее, чем когда либо ранее.

Если раньше языки программирования использовались лишь для создания программ для автоматизации вычислительных процессов, то на сегодняшний день они используются для решения более разнообразных задач.

Изучение истории языков программирования, их разнообразия и особенностей позволяет программисту сделать правильный выбор при выборе языка для решения определенной задачи.

Все многообразие языков программирования делят на различные классы в зависимости от решаемых ими задач. Было замечено, что в процессе развития языки программирования, входящие в один класс, сближаются между собой. Хотя само разнообразие классов увеличивается, т.к. увеличивается сфера задач, решаемых с помощью компьютерных технологий.

Цель нашей работы: рассмотреть современные языки программирования.

Для достижения поставленной цели нами были поставлены следующие задачи:

1) систематизировать основные этапы развития языков программирования и систем программирования;
2) выделить основные виды систем программирования;
3) рассмотреть основные компоненты системы программирования;
4) выявить требования к системам программирования;
5) выполнить обзор современных систем программирования.

При написании работы были проанализированы различные источники научно-технической литературы и статьи Интернет.

1 Основные этапы развития языков программирования

Под языком программирования (ЯП) понимают правила представления данных и записи алгоритмов их обработки, которые автоматически выполняются ЭВМ. В более абстрактном виде ЯП. является средством создания программных моделей объектов и явлений внешнего мира.

Первые ЭВМ, созданные человеком, имели небольшой набор команд и встроенных типов данных, но позволяли выполнять программы на машинном языке. Машинный язык (МЯ) – единственный язык, понятный ЭВМ. Он реализуется аппаратно: каждую команду выполняет некоторое электронное устройство. Программа на МЯ представляет собой последовательность команд и данных, заданных в цифровом виде.

Этот этап в развитии ЯП показал, что программирование является сложной проблемой, трудно поддающейся автоматизации, но именно программное обеспечение определяет в конечном счете эффективность применения ЭВМ. Поэтому на всех последующих этапах усилия направлялись на совершенствование интерфейса между программистом и ЭВМ – языка программирования.

Стремление программистов оперировать не цифрами, а символами, привело к созданию мнемонического языка программирования, который называют ассемблером. Этот язык имеет определенный синтаксис записи программ, в котором, в частности, цифровой код операции заменен мнемоническим кодом. Программа стала иметь более читаемую форму, но ее не понимала ЭВМ. Поэтому понадобилось создать специальную программу транслятор, который преобразует программу с языка ассемблера на МЯ. Эта проблема потребовала, в свою очередь, глубоких научных исследований и разработки различных теорий, например теорию формальных языков, которые легли в основу создания трансляторов. Практически любой класс ЭВМ имеет свой язык ассемблера. На сегодняшний день язык ассемблера используется для создания системных программ, использующих специфические аппаратные возможности данного класса ЭВМ.

Следующий этап характеризуется созданием языков высокого уровня (ЯВУ). Эти языки являются универсальными (на них можно создавать любые прикладные программы) и алгоритмически полными, имеют более широкий спектр типов данных и операций, поддерживают технологии программирования. На этих языках создается неисчислимое множество различных прикладных программ. Языки программирования высокого уровня делятся на несколько видов. (Приложение А)

Среди принципиальных отличий ЯВУ от языков низкого уровня выделяют следующее:

– использование переменных;

– возможность записи сложных выражений;

– расширяемость типов данных за счет конструирования новых типов из базовых;

– расширяемость набора операций за счет подключения библиотек подпрограмм;

– слабая зависимость от типа ЭВМ.

С усложнением ЯП усложняются и трансляторы для них. Теперь в набор инструментов программиста, кроме транслятора, входит текстовый редактор для ввода текста программ, отладчик для устранения ошибок, библиотекарь для создания библиотек программных модулей и множество других служебных программ. Все вместе это называется системой программирования. Наиболее яркими представителями ЯВУ являются FORTRAN, PL/1, Pascal, C, Basic, Ada.

Как можно заметить, было создано большое число языков одного класса. Каждый из разработчиков ЯВУ стремился создать самый лучший и самый универсальный язык, который позволял бы быстро получать самые эффективные, надежные и безошибочные программы. Однако в процессе этого поиска выяснилось, что дело не в самом языке, а в технологии его использования. Поэтому дальнейшее развитие языков стало определяться новыми технологиями программирования.

Одновременно с развитием универсальных ЯВУ стали развиваться проблемно-ориентированные ЯП, которые решали экономические задачи (COBOL), задачи реального времени (Modula-2, Ada), символьной обработки (Snobol), моделирования (GPSS, Simula, SmallTalk), численно-аналитические задачи (Analitic) и другие. Эти специализированные языки позволяли более адекватно описывать объекты и явления реального мира, приближая язык программирования к языку специалиста в проблемной области.

Другим направлением развития ЯП является создание языков сверхвысокого уровня (ЯСВУ). С помощью ЯП программист задает процедуру (алгоритм) получения результата по известным исходным данным, поэтому они называются процедурными ЯП. На ЯСВУ программист задает отношения между объектами в программе, например систему линейных уравнений, и определяет, что нужно найти, но не задает как получить результат. Такие языки еще называют непроцедурными, так как сама процедура поиска решения встроена в язык (в его интерпретатор). Такие языки используются, например, для решения задач искусственного интеллекта (Lisp, Prolog) и позволяют моделировать мыслительную деятельность человека в процессе поиска решений.

К непроцедурным языкам относят и языки запросов систем управления базами данных (QBE, SQL).

2 Классификация языков программирования

2.1 Машинно-ориентированные языки

Машинно-ориентированные языки – это языки, наборы операторов и изобразительные средства которых существенно зависят от особенностей ЭВМ (внутреннего языка, структуры памяти и т.д.). Машинно – ориентированные языки позволяют использовать все возможности и особенности Машинно – зависимых языков:

- высокое качество создаваемых программ (компактность и скорость выполнения);
- возможность использования конкретных аппаратных ресурсов;
- предсказуемость объектного кода и заказов памяти;
- для составления эффективных программ необходимо знать систему команд и особенности функционирования данной ЭВМ;
- трудоемкость процесса составления программ (особенно на машинных языках и ЯСК), плохо защищенного от появления ошибок;
- низкая скорость программирования;
- невозможность непосредственного использования программ, составленных на этих языках, на ЭВМ других типов.

Машинно-ориентированные языки по степени автоматического программирования подразделяются на классы.

2.1.1 Машинный язык

Каждый отдельный компьютер имеет свой определенный Машинный язык (далее МЯ), ему предписывают выполнение указываемых операций над определяемыми ими операндами, поэтому МЯ является командным. Однако, некоторые семейства ЭВМ (например, ЕС ЭВМ, IBM/370/ и др.) имеют

единый МЯ для ЭВМ разной мощности. В команде любого из них сообщается информация о местонахождении операндов и типе выполняемой операции.

В новых модулях ЭВМ намечается тенденция к повышению внутренних языков машинно-аппаратным путем реализовывать более сложные команды, приближающиеся по своим функциональным действиям к операторам алгоритмических языков программирования.

2.1.2 Языки Символического Кодирования

Продолжим рассказ о командных языках, Языки Символического Кодирования (далее ЯСК), так же, как и МЯ, являются командными. Однако коды операций и адреса в машинных командах, представляющие собой последовательность двоичных (во внутреннем коде) или восьмеричных (часто используемых при написании программ) цифр, в ЯСК заменены на символы (идентификаторы), форма написания которых помогает программисту легче запоминать смысловое содержание операции. Это обеспечивает существенное уменьшение числа ошибок при составлении программ.

Использование символических адресов – первый шаг к созданию ЯСК. Команды ЭВМ вместо истинных (физических) адресов содержат символические адреса. По результатам составленной программы определяется требуемое количество ячеек для хранения исходных промежуточных и результирующих значений. Назначение адресов, выполняемое отдельно от составления программы в символических адресах, может проводиться менее квалифицированным программистом или специальной программой, что в значительной степени облегчает труд программиста.

2.1.3 Автокоды

Есть также языки, включающие в себя все возможности ЯСК, посредством расширенного введения макрокоманд - они называются Автокоды.

В различных программах встречаются некоторые достаточно часто использующиеся командные последовательности, которые соответствуют определенным процедурам преобразования информации. Эффективная реализация таких процедур обеспечивается оформлением их в виде специальных макрокоманд и включением последних в язык программирования , доступный программисту. Макрокоманды переводятся в машинные команды двумя путями – расстановкой и генерированием. В постановочной системе содержатся «остовы» - серии команд, реализующих требуемую функцию, обозначенную макрокомандой. Макрокоманды обеспечивают передачу фактических параметров, которые в процессе трансляции вставляются в «остов» программы, превращая её в реальную машинную программу.

В системе с генерацией имеются специальные программы, анализирующие макрокоманду, которые определяют, какую функцию необходимо выполнить и формируют необходимую последовательность команд, реализующих данную функцию.

Обе указанных системы используют трансляторы с ЯСК и набор макрокоманд, которые также являются операторами автокода.

Развитые автокоды получили название Ассемблеры. Сервисные программы и пр., как правило, составлены на языках типа Ассемблер.

2.1.4 Макрос

Язык, являющийся средством для замены последовательности символов описывающих выполнение требуемых действий ЭВМ на более сжатую форму - называется Макрос (средство замены).

В основном, Макрос предназначен для того, чтобы сократить запись исходной программы. Компонент программного обеспечения,

обеспечивающий функционирование макросов, называется макропроцессором. На макропроцессор поступает макроопределяющий и исходный текст. Реакция макропроцессора на вызов-выдача выходного текста.

Макрос одинаково может работать, как с программами, так и с данными.

2.2 Машинно-независимые языки

Машинно-независимые языки – это средство описания алгоритмов решения задач и информации, подлежащей обработке [8,10,12]. Они удобны в использовании для широкого круга пользователей и не требуют от них знания особенностей организации функционирования ЭВМ и ВС.

Подобные языки получили название высокоуровневых языков программирования. Программы, составляемые на таких языках, представляют собой последовательности операторов, структурированные согласно правилам рассматривания языка(задачи, сегменты, блоки и т.д.). Операторы языка описывают действия, которые должна выполнять система после трансляции программы на МЯ.

Т.о., командные последовательности (процедуры, подпрограммы), часто используемые в машинных программах, представлены в высокоуровневых языках отдельными операторами. Программист получил возможность не расписывать в деталях вычислительный процесс на уровне машинных команд, а сосредоточиться на основных особенностях алгоритма.

2.2.1 Проблемно – ориентированные языки

С расширением областей применения вычислительной техники возникла необходимость формализовать представление постановки и решение новых классов задач. Необходимо было создать такие языки программирования, которые, используя в данной области обозначения и терминологию, позволили бы описывать требуемые алгоритмы решения для поставленных задач, ими стали проблемно – ориентированные языки. Эти

языки, языки ориентированные на решение определенных проблем, должны обеспечить программиста средствами, позволяющими коротко и четко формулировать задачу и получать результаты в требуемой форме.

Проблемных языков очень много, например:

- Фортран, Алгол – языки, созданные для решения математических задач;
- Simula, Слэнг - для моделирования;
- Лисп, Снобол – для работы со списочными структурами.

2.2.2 Универсальные языки

Универсальные языки были созданы для широкого круга задач: коммерческих, научных, моделирования и т.д. Первый универсальный язык был разработан фирмой IBM, ставший в последовательности языков Пл/1. Второй по мощности универсальный язык называется Алгол-68. Он позволяет работать с символами, разрядами, числами с фиксированной и плавающей запятой. Пл/1 имеет развитую систему операторов для управления форматами, для работы с полями переменной длины, с данными организованными в сложные структуры, и для эффективного использования каналов связи. Язык учитывает включенные во многие машины возможности прерывания и имеет соответствующие операторы. Предусмотрена возможность параллельного выполнение участков программ.

Программы в Пл/1 компилируются с помощью автоматических процедур. Язык использует многие свойства Фортрана, Алгола, Кобола. Однако он допускает не только динамическое, но и управляемое и статистическое распределения памяти.

2.2.3 Диалоговые языки

Появление новых технических возможностей поставило задачу перед системными программистами – создать программные средства,

обеспечивающие оперативное взаимодействие человека с ЭВМ их назвали диалоговыми языками [10,11].

Эти работы велись в двух направлениях. Создавались специальные управляющие языки для обеспечения оперативного воздействия на прохождение задач, которые составлялись на любых раннее неразработанных (не диалоговых) языках. Разрабатывались также языки, которые кроме целей управления обеспечивали бы описание алгоритмов решения задач.

Необходимость обеспечения оперативного взаимодействия с пользователем потребовала сохранения в памяти ЭВМ копии исходной программы даже после получения объектной программы в машинных кодах. При внесении изменений в программу с использованием диалогового языка система программирования с помощью специальных таблиц устанавливает взаимосвязь структур исходной и объектной программ. Это позволяет осуществить требуемые редакционные изменения в объектной программе.

Одним из примеров диалоговых языков является Бейсик.

Бейсик использует обозначения подобные обычным математическим выражениям. Многие операторы являются упрощенными вариантами операторов языка Фортран. Поэтому этот язык позволяет решать достаточно широкий круг задач.

2.2.4 Непроцедурные языки

Непроцедурные языки составляют группу языков, описывающих организацию данных, обрабатываемых по фиксированным алгоритмам (табличные языки и генераторы отчетов), и языков связи с операционными системами.

Позволяя четко описывать как задачу, так и необходимые для её решения действия, таблицы решений дают возможность в наглядной форме определить, какие условия должны быть выполнены прежде чем переходить к какому-либо действию. Одна таблица решений, описывающая некоторую

ситуацию, содержит все возможные блок-схемы реализаций алгоритмов решения.

Табличные методы легко осваиваются специалистами любых профессий.

Программы, составленные на табличном языке, удобно описывают сложные ситуации, возникающие при системном анализе.

3 Современные языки и системы программирования

3.1 Необходимость новых языков программирования

Начиная с 2000-х годов архитектура программирования начала меняться. Все больше вычислительных машин начали поставляться с несколькими процессорами, и даже отдельные процессоры имели более одного ядра. Этот сдвиг в характере вычислительного оборудования вызвал необходимость в языках программирования, которые в полной мере воспользовались новой архитектурой процессора. Языки должны были иметь возможность выполнять процессы одновременно и/или параллельно, чтобы максимизировать потенциал новых многоядерных процессоров. Параллелизм уже не был задумкой, его нужно было встроить в сам язык.

В последние годы также возрождался интерес к функциональному программированию, парадигме, которая пытается максимально устранить побочные эффекты. По сути, появление побочных эффектов оказалось проклятием в жизни современного программиста. Как правило, процесс отладки кода (один вид деятельности, который каждый программист боится) намного сложнее. В параллельной среде безопасность и неизменность данных становятся очень важными для разработчиков. Повреждения данных и / или расы должны быть предотвращены как можно больше.

И последнее, но не менее важное: современные машины продолжают расти очень мощно. Хотя в прошлом большое внимание уделялось скорости работы программ, этот фокус в последнее время угас. Вместо этого большое внимание переключилось на производительность программистов. В результате он платит за язык программирования за аккуратный и элегантный синтаксис, который легко писать и читать. Новый разработчик должен иметь возможность быстро подбирать его и работать с языком в кратчайшие сроки. Нужно иметь возможность играть с языком с самого начала с помощью интерактивного

цикла, обычно известного как REPL (Read-Eval-Print Loop), без необходимости проходить через утомительные процессы настройки и компиляции.

Чтобы оставаться актуальным в новом ландшафте программирования, были предприняты большие усилия для обновления старых языков, таких как Java и C ++, в соответствии с текущим положением дел. В Java, например, в выпуске Java 8 были добавлены *Lambda Expressions* и *API Streams* , а Java 9 теперь поставляется с REPL. Многопоточность библиотеки также были включены в языки, чтобы повысить их возможности параллелизма. Основным недостатком этих «старых» языков является то, что они никогда не строились с нуля для решения возникающих проблем в современном мире программирования. Попытка модифицировать новые функции на языках также не оказалась элегантным решением. Это не оставляет нам иного выбора, кроме как охватить новые языки программирования, которые были разработаны с нуля, чтобы решить некоторые сложные проблемы в современной разработке программного обеспечения.

Из-за описанных выше факторов (и многих других, не упомянутых в этой статье) было изобретено несколько языков программирования, чтобы попытаться решить некоторые (если не все) из этих проблем. Я называю эти языки « современными », потому что все они были выпущены в течение этого столетия. Как вы скоро поймете, большинство этих новых языков имеют много общего. Синтаксис некоторых из них очень похож.

Некоторые из общих функций, которыми пользуются большинство языков:

- переменные предпочитают быть *неизменяемыми* по умолчанию;
- тип *вывода*;
- большинство из них подчеркивают *безопасность типов*;
- большинство из них имеют *конечные* типы функция возврата;
- некоторые из них предлагают более простые способы создания нескольких процессов (или потоков), которые могут выполняться *одновременно*;

- некоторые из них предлагают более простые способы *межпроцессного взаимодействия* через каналы (или аналогичные примитивы);
- большинство из них подчеркивают *функциональный* стиль программирования (например, сопоставление образцов и ленивая оценка);
- большинство из них предлагают *REPL*;
- большинство из них являются *статически типизированными*;
- большинство из них имеют *чистый и элегантный синтаксис*, без лишних помех и многословия;

Ниже приведены, пожалуй, самые «видимые» современные языки программирования:

- Scala
- Golang (Go)
- Rust
- Kotlin
- Swift

Далее приводится обзор некоторых основных особенностей каждого языка.

3.1.1 Scala

Это язык, который дебютировал в первом десятилетии этого столетия и, возможно, является одним из «самых старых» современных языков программирования. Это продукт академических кругов, разработанный Мартином Одерским во время работы в EPFL в Швейцарии. Впервые выпущенный в 2004 году, Scala сочетает в себе объектно-ориентированные и функциональные парадигмы программирования. Приложения, написанные на Scala, работают поверх JVM, поэтому они легко переносятся на несколько платформ.

Объявление простой функции в Scala выглядит следующим образом:

```
def factorial(x: Int): Int = {
```

```
        // The body of the function goes here...
    }
```

Функция в Scala объявляется с использованием def ключевого слова (аналогичного Python и Ruby). За этим следует имя функции, в данном случае факториал , а затем список параметров в паре круглых скобок. Заметим , что параметр, в данном случае *x* , следует двоеточие (:) , а затем его тип данных (Int для целого числа). Если в списке имеется более одного параметра, они должны быть разделены запятой. После закрывающей скобки списка параметров функции у нас есть еще один двоеточие и Int после него. Это возвращаемый тип возвращаемой функции (более корректно называемый *типом результата* в Scala). Следуя типу результата функции, это знак равенства (=) и пару фигурных скобок, которые содержат тело функции. Знак равенства может показаться немного странным для тех, кто еще не знаком с Scala, но он имеет большой смысл с математической точки зрения. Содержимое в фигурных скобках можно рассматривать как математическое выражение, которое возвращает (или дает) результат в зависимости от определенного набора входных значений. Затем результат будет привязан к тому, что находится слева от знака равенства. Другими словами, когда вызывается *факторная* функция, она будет «*возвращать*» любой результат выражения внутри фигурных фигурных скобок.

Переменные в Scala объявляются с использованием либо ключевых слов (val или var аналогичных JavaScript). Переменные, объявленные с использованием ключевого слова *val* , неизменяемы и, следовательно, не переустанавливаются (это предпочтительный вариант в Scala). С другой стороны, переменные, объявленные с использованием ключевого слова *var,* могут быть переназначены. Если переменной присваивается начальное значение, то ее тип не обязательно должен быть объявлен явно, интерпретатор может легко «вывести» его тип из начального значения. Например:

```
val x = 20
```

В этом случае тип « *x* » будет считаться целым числом (Int). Если переменная не имеет начального значения, ее тип должен быть объявлен явно, например:

```
var x: Int
```

Важно отметить, что Scala требует инициализации переменной, независимо от того, намерены ли вы ее изменять или нет. Таким образом, для случая с использованием *var* выше, как объявление, так и инициализация переменной могут выполняться одновременно, например:

```
var x: Int = 20
```

Главный недостаток Scala - это его крутая кривая обучения.

3.1.2 Golang (Go)

Это язык, который был создан Google для собственного внутреннего использования (Google занимается высокораспределенными системами). Go особенно интересен, потому что один из его дизайнеров (Кен Томпсон) - фольклорный герой в области компьютерных наук. Он является создателем почтенной операционной системы UNIX , которую он разработал в начале 1970-х годов, работая в Bell Labs. Go был объявлен в 2009 году и с тех пор был выпущен как проект с открытым исходным кодом . Дизайнеры языка указали, что одной из их мотивов было их общее разочарование со сложностью, присущей C ++.

Go часто рекламируется как « C для 21-го века ». Это скомпилированный язык (с гораздо более быстрыми временами компиляции по сравнению с C / C ++) и имеет автоматическое управление памятью с помощью сборщика мусора .

Одной из сильных сторон Go является то, как она обрабатывает параллелизм с использованием goroutines и каналов .

Простая декларация функции в Go выглядит следующим образом:

```
func factorial(x int) int {
     // The body of the function goes here....

}
```

Функция в Go объявляется с использованием func ключевого слова. За этим следует имя функции и список параметров в паре круглых скобок. В этом случае факториальная функция принимает только один аргумент, целое число x . Функция также возвращает «целое число», как показано int ключевым словом после закрывающей круглой скобки списка аргументов. Тело функции заключено в пару фигурных скобок. Подобно Scala, возвращаемый тип функции появляется после имени функции (trailing return type), а не раньше, как в случае с C или Java.

Переменные в Go объявляются с использованием ключевого слова, var как показано ниже:

```
     var s string
s = "Welcome to Go"
```

Конечно, декларация и инициализация могут выполняться в одно и то же время:

```
var s string = "Welcome to Go"
```

Go также предлагает более короткий способ объявления переменных, как показано ниже:

```
s := "Welcome to Go"
```

В этом случае компилятор Go будет выводить ' s ' как a string.

Как уже упоминалось, параллелизм - это одна из областей, в которой Go сияет больше всего. Go использует goroutines и каналы для реализации параллелизма.

Ниже приведен простой пример:

```
 func main() {
c1 := make(chan string)
c2 := make(chan string)
```

```
    go func() {
        for {
            c1 <- "hello from channel 1"
        }
    }()

    go func() {
        for {
            c2 <- "hello from channel 2"
        }
    }()

    go func() {
        for {
            select {
                case msg1 := <- c1:
                    fmt.Println(mg1)
                case msg2 := <- c2:
                    fmt.Println(mg2)
                default:
                    fmt.Println("Nothing is ready yet.")
            }
        }
    }()

    var input string
    fmt.Scanln(&input)
}
```

Goroutine объявляется с помощью go ключевого слова. Горутины - это потоковые объекты, которые могут выполняться одновременно. Горутины общаются по каналам .

Тип канала указывается ключевым словом, chan за которым следует тип «трафика», который будет обрабатывать канал, как показано ниже:

```
c1 := make(chan string)
```

В этом случае мы используем встроенную make функцию для создания канала c1 , который может обрабатывать трафик string типа.

Оператор левой стрелки (<-) используется для отправки (и приема) сообщений по каналу, как показано ниже:

```
c1 <- "hello from channel 1"
```

В этом случае строковое сообщение « привет от канала 1 » является « отправить » на канал c1 .

Конструкция приема показана ниже:

```
msg1 := <- c1
```

Вышеприведенный фрагмент кода может быть интерпретирован как:

« Получать сообщение от канала c1 и хранить его в msg1 ».

Блок выбора выбирает первый канал, который готов, и получает от него сообщение. Если более одного из каналов готовы, тогда он случайным образом выбирает, какой из них будет получать. Если ни один из каналов не готов, оператор ожидает, пока канал станет доступным (процесс, обычно известный как блокирование). Но если предоставлен случай по умолчанию , тогда он будет выполнен вместо этого.

Самым известным проектом, реализованным с помощью Go, является, возможно, Docker - программная технология для создания контейнеров .

3.1.3 Rust

Rust - это язык системного программирования, который был создан Mozilla Research как более безопасная и эффективная альтернатива C / C ++. Он был официально выпущен в качестве проекта с открытым исходным кодом еще в 2010 году. Ведущим разработчиком этого языка был Грейдон Хоар , который с тех пор переехал в Apple и теперь работает как часть команды Swift (обсуждается позже).

Заявленные цели языка программирования Rust:

- безопасности
- скорость
- совпадение

Из этих трех, Rust использует безопасность наиболее серьезно. Он обеспечивает безопасность посредством одной из самых уникальных (и сложных) концепций: права собственности.

Простое объявление функции в Rust выглядит так:

```
fn factorial(x: i32) -> i32 {
    // The body of the function goes here....
}
```

Функция в Rust объявляется с использованием ключевого слова fn. За этим следует имя функции и список аргументов в паре круглых скобок. В приведенном выше случае у нас есть только один аргумент x , который представляет собой 32-разрядное integer число со знаком. Обратите внимание, что тип аргумента функции должен быть объявлен; и если есть несколько аргументов, они должны быть разделены запятыми.

Функция возвращает другое 32-разрядное целое число со знаком, указанное указателем правой стрелки (->) и i32 после него.

Переменные в Rust неизменяемы по умолчанию (безопасность берется очень серьезно в Rust!), И они объявляются с использованием ключевого слова let, как показано ниже:

```
let x = 7;
```

Вышеприведенный код объявляет локальную переменную и присваивает ей значение 7 . Фрагмент кода официально называется привязкой переменной в Rust. В этом случае 7 « привязано » к переменной x. Компилятор Rust выберет тип x as i32.

Идея переменных привязок позволяет сопоставить шаблоны, как показано ниже:

```
let (x, y) = (7, 13);
```

В этом случае 7 будет привязано к x и 13 к y .

Тип переменной может быть явно объявлен, как показано ниже:

```
let x: i32 = 7;
```

Если переменная должна измениться после ее инициализации, мы должны добавить ключевое слово mut(изменяемое) следующим образом:

```
let mut x = 7;
```

В отличие от большинства других рассмотренных языков, Rust требует точку с запятой в качестве терминатора утверждения (хотя для этого правила существует несколько исключений).

Будучи системным языком, Rust предлагает собственные примитивы параллелизма на уровне потока. Он обещает « параллелизм без гонок данных ».

Темы в Rust создаются с использованием следующего синтаксиса:

```
      let thread = std::thread::spawn(|| {
        println!("Graydon");
});
```

Чтобы создать новый поток, мы используем spawn()функцию из стандартной библиотеки потоков: std :: thread .

spawn() Функция принимает в качестве аргумента замыкания , которое в этом случае печатает string.

Следует обратить внимание на bang (!) В конце println . Это означает, что это macro вызов, а не обычный вызов функции.

Как и в Go, потоки в Rust взаимодействуют через каналы . Это иллюстрируется следующим примером:

```
      use std::thread;
use std::sync::mpsc;
      fn main() {
   let (tx, rx) = mpsc::channel();

   for _ in 0..10 {
        let tx = tx.clone();

        thread::spawn(move || {
             let result = 77u32;

             tx.send(result);
        });
   }

   rx.recv().ok().expect("could not receive the result.");
}
```

Сначала идет импорт стандартных потоков и sync библиотек synchronization ( ).

Канал создается путем вызова channel() метода из пакета std :: sync :: mpsc , как показано ниже:

```
      let (tx, rx) = mpsc::channel();
```

В этом случае канал связан с кортежем арностью 2 (Arity относится к числу аргументов операция занимает).

Инициалы в mpsc означают « множественный производитель, один потребитель» . Это означает, что « отправляющий » конец канала (в данном случае tx) может быть клонирован (копии его могут быть сделаны) и использоваться таким количеством потоки, как вы хотели бы присвоить значения; но « приемный » конец канала (в данном случае, rx) не может быть клонирован, поэтому только один поток может извлекать значения из очереди.

После создания потока он вычисляет результат и отправляет его по каналу через tx , как показано ниже:

```
tx.send(result);
```

В примере используется move замыкание при создании потока. Это обеспечивает право собственности , предоставляя закрытию собственный стек стека .

Если rx получает результат , expect метод возвращает значение результата, иначе он возвращает ошибку Err(e), со строковым сообщением: « не смог получить результат ».

Стандартный пакет синхронизации Rust (std :: sync) предлагает еще два типа, которые полезны для обеспечения безопасности в многопоточной среде: Mutex<T> и Arc<T>. Чтобы использовать их, их необходимо импортировать, как показано ниже:

```
use std::sync::Mutex;
use std::sync::Arc;
```

Вышеупомянутый импорт также может быть выполнен с использованием только одной строки, например:

```
use std::sync::{Mutex, Arc};
```

Тип Mutex предлагает блокировки через свой lockметод. Это гарантирует, что два потока не могут требовать владения одной и той же частью данных одновременно.

Инициалы в Arc обозначают « Atomic reference count ». Тип Arc гарантирует, что совместное использование неизменяемых данных является потокобезопасным . Он пытается избежать расчётов данных между несколькими потоками.

Наконец, Rust предлагает надежный менеджер пакетов под названием Cargo . Это помогает в управлении внешними кодовыми библиотеками Rust (теми, которые не входят в стандартную библиотеку), известными как « ящики » сообщества Rust.

Следующий фрагмент кода показывает, как вы можете использовать вызванный библиотечный ящик rand, который предлагает функциональность случайных чисел:

```
extern crate rand;
extern
```

Ключевое слово указывает на то, что это внешняя библиотека (обрешетка).

Другая область, в которой Rust отличается, - это поддержка FFI(Интерфейс внешних функций). Это означает, что его можно легко встроить в другие языки.

С точки зрения его кривой обучения, Rust лежал бы чуть ниже Scala .

Следует упомянуть, что согласно опросу разработчиков стека Overflow , Rust был самым любимым языком программирования в течение двух лет подряд (2016 и 2017).

3.1.4 Kotlin

До недавнего времени Котлин был едва известен в рамках большого сообщества разработчиков. Язык, названный в честь острова Котлинвблизи Санкт-Петербурга в России, был создан компанией JetBrains , компанией популярных продуктов, таких как IntelliJ IDEA, PhpStorm и PyCharm . Он был

официально выпущен в 2011 году, и он работает поверх виртуальной машины Java (аналогично Scala).

Kotlin получил очень большой импульс, когда в выпуске ввода-вывода Google в 2017 году было объявлено, что языку была оказана первоклассная поддержка на мобильной платформе Android (став только третьим языком, которому будет предоставлен такой статус, после Java и C ++). Новость была получена с большим волнением со стороны сообщества разработчиков Android. В результате интерес к этому языку резко возрос в течение последних нескольких месяцев после объявления. Кроме того, это может рассматриваться как стратегический шаг Google, поскольку он был вовлечен в юридическую борьбу с Oracle за использование определенных классов Java для Android.

Одна из заявленных целей Kotlin заключается в том, чтобы полностью взаимодействовать с Java и работать так же быстро, как собственный код Java.

Простая декларация функции в Kotlin выглядит так:

```
fun factorial(x: Int): Int {
       // The body of the function goes here...
}
```

Этот блок кода очень похож на код Scala, показанный ранее, и поэтому он не требует большого объяснения. Единственное, что нужно отметить здесь, это то, что в Котлине функция объявляется с использованием ключевого слова fun. В этом случае функция принимает один аргумент типа Intи возвращает Int также.

Как и в Scala, переменные в Kotlin объявляются с использованием либо val или var ключевых слов, как показано ниже:

```
val x: Int = 3
var y = 7
```

В этом случае x является неизменной переменной типа Int, а y изменчива. Тип y определяется как Int.

По сравнению с тремя языками, рассмотренными выше, я считаю, что реализация параллелизма в Котлине является наименее надежной.

Kotlin использует сопрограммы для реализации параллелизма (хотя в последний раз, когда я проверял, он все еще находился на экспериментальной фазе). Он применяет концепцию подвески , что означает, что вычисления могут быть приостановлены без блокировки рабочей нити.

Вот несколько причин, почему можно полностью переключиться на Kotlin:

– Kotlin на 100% совместим с Java . Вы можете буквально продолжить работу над старыми Java-проектами, используя Kotlin. Все ваши любимые фреймворки Java по-прежнему доступны , и любые рамки, которые вы будете писать в Kotlin, легко принимаются вашим упрямым другом, любящим Java.

– Котлин - не какой-то странный язык, родившийся в академических кругах. Его синтаксис знакомы любому программисту, поступающему из домена ООП , и его можно более или менее понимать с самого начала. Конечно, есть *некоторые* отличия от Java, такие как переработанные конструкторы или val var объявления переменных.

– Компилятор Kotlin отслеживает логические и автоматические типы, если это возможно , что означает, что больше никаких instanceof проверок не следует явным приведениям.

– Нет необходимости определять несколько подобных методов с различными аргументами.

– В сочетании с аргументами по умолчанию именованные аргументы устраняют необходимость в сборщиках и др.

3.1.5 Swift

Swift - мощный и интуитивно понятный язык программирования для macOS, iOS, watchOS и tvOS. Написание кода Swift является интерактивным и забавным, синтаксис является сжатым, но выразительным, а Swift включает в себя современные функции, которые разработчики любят. Быстрый код

безопасен по дизайну, но также производит программное обеспечение, которое работает молниеносно.

Созданный в Apple, Swift предназначен для более безопасной и более сжатой альтернативы *Objective-C* для разработки приложений в экосистеме Apple. Язык был представлен на Всемирной конференции разработчиков Apple в 2014 году и выпущен в качестве проекта с открытым исходным кодом год спустя. Его ведущим дизайнером был Крис Лэттнер (который также является одним из оригинальных авторов проекта LLVM).

Swift является результатом последних исследований по языкам программирования в сочетании с многолетним опытом создания платформ Apple. Именованные параметры, перенесенные из Objective-C, выражаются в чистом синтаксисе, который упрощает чтение и обслуживание API в Swift. Выведенные типы делают код более чистым и менее подверженным ошибкам, тогда как модули исключают заголовки и предоставляют пространства имен. Память управляется автоматически, и даже не нужно вводить полуколоны. Эти перспективные концепции приводят к легкому и интересному языку.

Большая часть кода Swift немного похожа на код, написанный в Rust.

Простая декларация функции в Swift написана, как показано ниже:

```
func factorial(x: Int) -> Int {
        // The body of the function goes here...
}
```

Функция в Swift объявляется с использованием ключевого слова func(похожего на **Go**). В этом конкретном случае функция имеет один аргумент типа Intи имеет Intкак возвращаемый тип.

Свифт проводит четкое различие между неизменяемыми и изменяемыми переменными. Неизменяемые переменные называются *константами,* а их изменчивые аналоги называются *переменными* (как и ожидалось).

Константы объявляются с использованием let ключевого слова, а переменные объявляются с использованием var ключевого слова. Простая иллюстрация показана ниже:

```
let str: String = "Welcome to Swift Programming"
var x = 10
```

Первая строка объявляет константу *str* типа String, а вторая строка объявляет переменную *x* , тип которой вызывается как Int.

К сожалению, Swift очень медленно применяет функции параллелизма на уровне языка (хотя, похоже, некоторый прогресс начинается с Swift 3).

Возможно, самой знаковой особенностью Swift является наличие *Swift Playgrounds* , в котором есть своего рода песочница, в которой код Swift может исполняться « *на лету* ». Еще более интересным является тот факт, что игровые площадки также работают на iPads!

У Swift есть много других возможностей, чтобы сделать код более выразительным:

– Закрытие унифицировано с указателями функций
– Кортежи и множественные возвращаемые значения
– Дженерики
– Быстрая и краткая итерация по диапазону или коллекции
– Структуры, поддерживающие методы, расширения и протоколы
– Функциональные шаблоны программирования, например, карта и фильтр
– Собственная обработка ошибок с помощью try / catch / throw

Swift исключает целые классы небезопасного кода. Переменные всегда инициализируются перед использованием, массивы и целые числа проверяются на переполнение, а управление памятью происходит автоматически. Синтаксис настраивается, чтобы упростить определение

вашего намерения - например, простые трехсимвольные ключевые слова определяют переменную (var) или константу (let).

Еще одна особенность безопасности заключается в том, что по умолчанию объекты Swift никогда не могут быть nil . Фактически, компилятор Swift остановит вас от попытки создания или использования объекта nil с ошибкой времени компиляции. Это делает код записи намного чище и безопаснее, и предотвращает огромную категорию сбоев во время выполнения в ваших приложениях. Однако есть случаи, когда nil является допустимым и подходящим. В этих ситуациях Swift имеет инновационную функцию, называемую опциональными . Необязательный может содержать ноль, но синтаксис Swift заставляет вас безопасно справляться с ним, используя ? синтаксис, чтобы указать компилятору, что вы понимаете поведение и будете безопасно обращаться с ним.

Есть замечательный ресурс для изучения Swift, - это бесплатное руководство, предлагаемое Apple, которое можно загрузить как файл ePub. Этот документ часто известен как « Быстрый язык программирования ».

Заключение

Компьютерные программы часто описываются как «наборы инструкций», а многие языки компьютеров считаются просто синтаксисом и лексикой для предоставления этих инструкций.

С этой точки зрения разные языки программирования могут иметь разные грамматики или разные словари. Каждый из них может обрабатывать полуколоны определенным образом или требовать капитализации, но они все равно похожи на все это.

Реальность программирования намного сложнее.

Большинство действительно «больших» идей в программировании были разработаны в 1950-х и 60-х годах. С тех пор появилось много новых языков, но ни один из них не представляет собой по-настоящему новый подход к логике и вычислению.

Развитие новых языков программирования за последние несколько десятилетий сфокусировалось на опыте разработчиков. Это может означать попытку включения кода, который легче писать (движущая сила Ruby) или более легкого для чтения (Python), или создания определенных типов логических структур и способов решения проблем более интуитивно понятными.

Некоторые языки были разработаны для решения конкретных проблем программирования (например, PHP и SASS), для управления некоторыми типами систем (SQL) или для работы в определенной среде или платформе (Java и JavaScript). Был разработан ряд языков с целью помочь новичкам научиться программированию (BASIC и Scratch являются классическими примерами).

Поскольку теории и практики в области языкового дизайна (в основном) опираются на широко признанную ортодоксию, большая часть новой и интересной работы по разработке практики программирования в настоящее время сосредоточена вокруг архитектуры системы.

Трудно сказать, в каком именно направлении идет программирование. В краткосрочной перспективе мы, вероятно, ожидаем большего ускорения тенденций, которые мы уже испытываем:

- Большие данные
- Виртуализация
- "Интернет вещей"

Но в долгосрочной перспективе, как известно, трудно сделать точные прогнозы. Квантовые вычисления могут привести к совершенно новой парадигме компьютерного программирования; компьютеры могут научиться программировать себя, приводя к Сингулярности и концу человеческой эры; мы можем узнать, как использовать программирование для имитации биологического интеллекта, приводящего к трансгуманизму. Или мы могли бы просто выяснить, как сделать наши телефоны меньше.

Независимо от будущего, становится все более и более очевидным, что программирование - возможность читать и писать код на нескольких языках - становится новой бизнес-грамотностью. Знакомство с концепциями программирования и логикой компьютерных систем и архитектуры быстро становится таким же важным, как основные бизнес-навыки, такие как продажи, маркетинг и дизайн.

Список использованных источников

1. Алексеев В.Е. и др. Вычислительная техника и программирование. Практикум по программированию. - М.: ВШ, 2016, 200 с.
2. Бондарев В. - М., Рублинецкий В.И., Качко Е.Г. Основы программирования. - Харьков: Фолио, Ростов н/Д: Феникс, 2015. - 368 с.
3. Вирт Н. Алгоритмы и структуры данных. - М.: Мир, 2013, 406 с.
4. Исакова С., Жемеров Д. Kotlin в действии. - М.: ДМК Пресс, 2017. – 402 с.
5. Кнут, Д. Искусство программирования для ЭВМ; М.: Мир - Москва, 2015. - 569 с.
6. Кьоу, Дж.; Джеанини, М. Объектно-ориентированное программирование. Просто и понятно; СПб: Питер - Москва, 2015. - 238 с.
7. Крюков Е.А. Язык программирования Go. Руководство 2016; Издательство: Accent Graphics communications, 2016. - 358 с.
8. Саммерфильд Марк. Программирование на Go. Разработка приложений XXI века: пер. с англ.: Кисвелев А.Н. – М.: ДМК Пресс, 2016. – 580 с.
9. Джим Блэнди, Джейсон Орендорф. Программирование на языке Rust; – М.: ДМК Пресс, 2018. – 550 с.
10. Одерски Мартин, Спун Лекс, Веннерс Билл. Scala. Профессиональное программирование; "Издательский дом "Питер", 2018. - 685 с.
11. Орлов С.А. Теория и практика языков программирования. Учебник для вузов. 2-е изд. Стандарт 3-го поколения; "Издательский дом "Питер", 2017. - 688 с.
12. Усов В. Swift. Основы разработки приложений под iOS. — СПб.: Питер, 2016. — 304 с.: ил.
13. Якунин, Ю. Ю. Технологии разработки программного обеспечения. Версия 1.0 [Текст] : / Ю. Ю. Якунин.- Красноярск , 2015. – 225 с.

14. Язык программирования Kotlin / Сергей Пименов — К. : «Агентство «IPIO», 2017. — 304 с.

15. Восхождение современных языков программирования [Электронный ресурс], режим доступа: https://medium.com/the-andela-way/the-rise-of-modern-programming-languages-c923a2b914fc (дата обращения: 25.06.2018).

16. Технология программирования. Основные понятия и подходы [Электронный ресурс], режим доступа: http://www.arctic-cooler.com/programming/1/comptechnology0.htm (дата обращения: 25.06.2018).

17. История развития языков программирования [Электронный ресурс], режим доступа: http://life-prog.ru/view_articles.php?id=41(дата обращения: 25.06.2018).

18. История создания языков программирования [Электронный ресурс], режим доступа: http://www.shapovalov.org/publ/7-1-0-77 (дата обращения: 25.06.2018).

19. Rust на примерах. Часть 1 [Электронный ресурс], режим доступа: https://habr.com/post/232829/ (дата обращения: 25.06.2018).

20. Учебник по языку программирования Swift на Русском [Электронный ресурс], режим доступа: http://imania.do.am/swift/Swift_rus_imania.do.am.pdf (дата обращения: 25.06.2018).