1. Лексика, Синтаксис, Семантика

Лексика — правила «правописания слов» программы, таких как идентификаторы, константы, служебные слова, комментарии. Лексический анализ разбивает текст программы на указанные элементы. Особенность любой лексики — ее элементы представляют собой регулярные линейные последовательности символов.

Например, идентификатор — это произвольная последовательность букв, цифр и символа «\_», начинающаяся с буквы или «\_». Лексика ЯП анализируется и интерпретируется на фазе лексического анализа при трансляции.

Синтаксис языка программирования — набор правил, описывающий комбинации символов алфавита, считающиеся правильно структурированной программой (документом) или её фрагментом. Синтаксису языка противопоставляется его семантика. Синтаксис языка описывает «чистый» язык, в то же время семантика приписывает значения (действия) различным синтаксическим конструкциям.

Каждый язык программирования имеет синтаксическое описание как часть грамматики. Синтаксис языка можно описать, например, с помощью правил Бэкуса — Наура.

Форма Бэкуса — Наура (сокр. БНФ, Бэкуса — Наура форма) — формальная система описания синтаксиса, в которой одни синтаксические категории последовательно определяются через другие категории. БНФ используется для описания контекстно-свободных формальных грамматик. Существует расширенная форма Бэкуса — Наура, отличающаяся лишь более ёмкими конструкциями.

БНФ-конструкция состоит из нескольких предложений вида:

<определяемый символ> ::=<посл.1> | <посл.2> | . . . | <посл.n>

Семантика в программировании — дисциплина, изучающая формализации значений конструкций языков программирования посредством построения их формальных математических моделей. В качестве инструментов построения таких моделей могут использоваться различные средства, например, математическая логика, λ-исчисление, теория множеств, теория категорий, теория моделей, универсальная алгебра. Формализация семантики языка программирования может использоваться как для описания языка, определения свойств языка, так и для целей формальной верификации программ на этом языке программирования.

1. Статическое окружение

Статическое окружение для переменной — это когда связь, установленная в более внешнем вызове, остается в силе на протяжении всего выполнения программы для этой переменной.

Впервые происходит перед запуском программы (на этапе компиляции) и остается неизбежным на протяжении её выполнения.

Статическое окружение формируется для того, чтобы проверить правильность построения программы.

* val a = 25                                   [(a, int)]
* val b = 71                                   [(b, int), (a, int)]
* val c = real a + 25.3                   [(c, real), (b, int), (a, int)]
* fun f (x, y) = x\* 5 + y                  [(f (int, int) -> int), (c, real), (b, int), (a, int)]
* val d = [a, f (b, a - 7)]                 [(d, int list), (f (int, int) -> int), (c, real), (b, int), (a, int)]
* val a = "afcd"                            [(a, string), (d, int list), (f (int, int) -> int), (c, real), (b, int), (a, int)]
* val c = hd d                                [(c, int),(a, string), (d, int list), (f (int, int) -> int), (c, real), (b, int), (a, int)]

Если все проверки на соответствие типов пройдены и программа выполняется корректно, то статическое окружение больше не нужно и далее используется динамическое окружение.

1. Функции как объекты первого класса

Для начала нужно понять, что подразумевается под *функциями первого класса*

Под этим определением подразумевается, что функцию можно передать в качестве аргумента другой функции, вернуть её как результат другой функции, присвоить её переменной или сохранить в структуре данных. Стоит отметить, что в языке Standart ML функции представлены в виде *замыканий,*т.е. функций, ссылающихся на свободные переменные в своей области видимости.

Например, функция map (функция высшего порядка) принимает в качестве своих аргументов функцию и список и возвращается список, после применения функции к каждому элементу списка. Для наличия этой функции необходимо, чтобы функция могла передаваться как аргумент

**Пример:** *var a = fn x => x \* x*

В примере мы присвоили переменной значение функции возведения в квадрат числа x

1. Формы результата выражения на Haskell

1) thunk - невычисленное выражение, которое возможно ещё предстоит вычислить. Пример: thunk - nat = 1 : map (\x->x+1) nat. Ленивая система вычислений не вычисляет thunk, если это не нужно.Например при работе с бесконечными списками, мы не можем вычислить весь список  т.к. это бесконечный процесс, но можем вычислить и изменить наш список до то части (включаю её), которая нам необходима. Например, take 5 (drop 1000 [1,..]) результат - 1001 1002 1003 1004 1005

2) слабая головная нормальная форма - выражение вычисленное до значения, которое является вызовом конструктора от невычисленного или частично вычисленных аргументов. Примеры выражений в слабой головной форме: (1 + 1, 2 + 2)       \x -> 2 + 2            'h' : ("e" ++ "llo")

3) нормальная форма - полностью вычисленное выражение и никакое подвыражение не может быть вычисленно дальше (т.е. не содержит не оцененных разрывов). Примеры выражений в нормальной форме: 42                         (2, "hello")                \x -> (x + 1)

4) дно - выражение, которое не может быть вычисленно. Например 1/0.

5.Полиморфные функции

В [языках программирования](https://ru.wikipedia.org/wiki/%D0%AF%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F) и [теории типов](https://ru.wikipedia.org/wiki/%D0%A2%D0%B5%D0%BE%D1%80%D0%B8%D1%8F_%D1%82%D0%B8%D0%BF%D0%BE%D0%B2) **полиморфизмом** называется способность [функции](https://ru.wikipedia.org/wiki/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) обрабатывать данные разных [типов](https://ru.wikipedia.org/wiki/%D0%A2%D0%B8%D0%BF_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85).

Существует несколько разновидностей полиморфизма. Две принципиально различных из них были это [параметрический полиморфизм](https://ru.wikipedia.org/wiki/%D0%9F%D0%B0%D1%80%D0%B0%D0%BC%D0%B5%D1%82%D1%80%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D0%BF%D0%BE%D0%BB%D0%B8%D0%BC%D0%BE%D1%80%D1%84%D0%B8%D0%B7%D0%BC) и [ad hoc полиморфизм](https://ru.wikipedia.org/wiki/Ad_hoc_%D0%BF%D0%BE%D0%BB%D0%B8%D0%BC%D0%BE%D1%80%D1%84%D0%B8%D0%B7%D0%BC" \o "Ad hoc полиморфизм), причём первая является истинной формой, а вторая - мнимой; прочие формы являются их подвидами или сочетаниями.  
Параметрический полиморфизм подразумевает исполнение **одного и того же** кода для всех допустимых типов аргументов, тогда как ad hoc полиморфизм подразумевает исполнение потенциально **разного** кода для каждого типа или подтипа аргумента.

Принципиальная возможность для одного и того же кода обрабатывать данные разных типов определяется свойствами [системы типов](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0_%D1%82%D0%B8%D0%BF%D0%BE%D0%B2) языка. С этой точки зрения различают статическую неполиморфную типизацию (потомки [Алгола](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D0%BB) и [BCPL](https://ru.wikipedia.org/wiki/BCPL)), динамическую типизацию (потомки [Lisp](https://ru.wikipedia.org/wiki/Lisp" \o "Lisp), [Smalltalk](https://ru.wikipedia.org/wiki/Smalltalk" \o "Smalltalk), [APL](https://ru.wikipedia.org/wiki/APL_(%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F))) и статическую [полиморфную типизацию](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%BB%D0%B8%D0%BC%D0%BE%D1%80%D1%84%D0%B8%D0%B7%D0%BC_%D1%82%D0%B8%D0%BF%D0%BE%D0%B2) (потомки [ML](https://ru.wikipedia.org/wiki/ML)). Использование ad hoc полиморфизма наиболее характерно для неполиморфной типизации. Параметрический полиморфизм и динамическая типизация намного существеннее, чем ad hoc полиморфизм, повышают коэффициент [повторного использования кода](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%B2%D1%82%D0%BE%D1%80%D0%BD%D0%BE%D0%B5_%D0%B8%D1%81%D0%BF%D0%BE%D0%BB%D1%8C%D0%B7%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_%D0%BA%D0%BE%D0%B4%D0%B0), поскольку определённая единственный раз функция реализует без дублирования заданное поведение для бесконечного множества вновь определяемых типов, удовлетворяющих требуемым в функции условиям. С другой стороны, временами возникает необходимость обеспечить различное поведение функции в зависимости от типа параметра, и тогда необходимым оказывается специальный полиморфизм.

[Параметрический полиморфизм](https://ru.wikipedia.org/wiki/%D0%9F%D0%B0%D1%80%D0%B0%D0%BC%D0%B5%D1%82%D1%80%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D0%BF%D0%BE%D0%BB%D0%B8%D0%BC%D0%BE%D1%80%D1%84%D0%B8%D0%B7%D0%BC)[>>>](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%BB%D0%B8%D0%BC%D0%BE%D1%80%D1%84%D0%B8%D0%B7%D0%BC_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)#%D0%BF%D0%B0%D1%80%D0%B0%D0%BC%D0%B5%D1%82%D1%80%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D0%BF%D0%BE%D0%BB%D0%B8%D0%BC%D0%BE%D1%80%D1%84%D0%B8%D0%B7%D0%BC) является синонимом **абстракции типа**. Он повсеместно используется в [функциональном программировании](https://ru.wikipedia.org/wiki/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%BE%D0%BD%D0%B0%D0%BB%D1%8C%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5), где он обычно обозначается просто как «полиморфизм».

В сообществе [объектно-ориентированного программирования](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) под термином «полиморфизм» обычно подразумевают [наследование](https://ru.wikipedia.org/wiki/%D0%9D%D0%B0%D1%81%D0%BB%D0%B5%D0%B4%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)), а использование параметрического полиморфизма называют [обобщённым программирование](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D0%BE%D0%B1%D1%89%D1%91%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)м, или иногда «статическим полиморфизмом».

Если параметру функции сопоставлено более одного типа, то такая функция называется **полиморфной**. Разумеется, с каждым фактическим значением может быть связан лишь один тип, но полиморфная функция рассматривает параметры на основе внешних свойств, а не их собственной организации и содержания. Стрэчи назвал такую возможность «*параметрическим полиморфизмом*».

В дальнейшем классификацию уточнил [Лука Карделли](https://ru.wikipedia.org/w/index.php?title=%D0%9A%D0%B0%D1%80%D0%B4%D0%B5%D0%BB%D0%BB%D0%B8,_%D0%9B%D1%83%D0%BA%D0%B0&action=edit&redlink=1), выделив четыре разновидности полиморфизма:

* универсальный
  + [параметрический](https://ru.wikipedia.org/wiki/%D0%9F%D0%B0%D1%80%D0%B0%D0%BC%D0%B5%D1%82%D1%80%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D0%BF%D0%BE%D0%BB%D0%B8%D0%BC%D0%BE%D1%80%D1%84%D0%B8%D0%B7%D0%BC)
  + включения (или [подтипов](https://ru.wikipedia.org/w/index.php?title=%D0%92%D1%8B%D0%B4%D0%B5%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5_%D0%BF%D0%BE%D0%B4%D1%82%D0%B8%D0%BF%D0%BE%D0%B2_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85&action=edit&redlink=1))
* ad hoc
  + [перегрузка](https://ru.wikipedia.org/wiki/%D0%9F%D0%B5%D1%80%D0%B5%D0%B3%D1%80%D1%83%D0%B7%D0%BA%D0%B0_%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%B9)
  + [приведение типов](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B8%D0%B2%D0%B5%D0%B4%D0%B5%D0%BD%D0%B8%D0%B5_%D1%82%D0%B8%D0%BF%D0%BE%D0%B2)

Параметрический полиморфизм позволяет определять функцию или тип данных обобщённо, так что значения обрабатываются идентично вне зависимости от их типа. Параметрически полиморфная функция использует аргументы на основе поведения, а не значения, апеллируя лишь к необходимым ей свойствам аргументов, что делает её применимой в любом контексте, где тип объекта удовлетворяет заданным требованиям поведения.

Развитые [системы типов](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0_%D1%82%D0%B8%D0%BF%D0%BE%D0%B2) (такие как [Хиндли — Милнер](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0_%D1%82%D0%B8%D0%BF%D0%BE%D0%B2_%D0%A5%D0%B8%D0%BD%D0%B4%D0%BB%D0%B8_%E2%80%94_%D0%9C%D0%B8%D0%BB%D0%BD%D0%B5%D1%80%D0%B0" \o "Система типов Хиндли — Милнера)) предоставляют механизмы для определения [полиморфных типов](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%BB%D0%B8%D0%BC%D0%BE%D1%80%D1%84%D0%B8%D0%B7%D0%BC_%D1%82%D0%B8%D0%BF%D0%BE%D0%B2), что делает использование полиморфных функций более удобным и обеспечивает статическую [типобезопасность](https://ru.wikipedia.org/wiki/%D0%A2%D0%B8%D0%BF%D0%BE%D0%B1%D0%B5%D0%B7%D0%BE%D0%BF%D0%B0%D1%81%D0%BD%D0%BE%D1%81%D1%82%D1%8C" \o "Типобезопасность). Такие системы являются системами типов второго порядка, добавляющими к системам типов первого порядка (используемым в большинстве [процедурных языков](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D0%B4%D1%83%D1%80%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) параметризацию типов (посредством [ти́повой переменной](https://ru.wikipedia.org/wiki/%D0%9F%D0%B5%D1%80%D0%B5%D0%BC%D0%B5%D0%BD%D0%BD%D0%B0%D1%8F_%D1%82%D0%B8%D0%BF%D0%B0" \o "Переменная типа)) и [абстракцию типов](https://ru.wikipedia.org/wiki/%D0%90%D0%B1%D1%81%D1%82%D1%80%D0%B0%D0%BA%D1%82%D0%BD%D1%8B%D0%B9_%D1%82%D0%B8%D0%BF_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85) (посредством [экзистенциальной квантификации](https://ru.wikipedia.org/wiki/%D0%9F%D0%B5%D1%80%D0%B5%D0%BC%D0%B5%D0%BD%D0%BD%D0%B0%D1%8F_%D1%82%D0%B8%D0%BF%D0%B0#%D0%A1%D0%B2%D1%8F%D0%B7%D1%8B%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_%D0%B8_%D0%BA%D0%B2%D0%B0%D0%BD%D1%82%D0%B8%D1%84%D0%B8%D0%BA%D0%B0%D1%86%D0%B8%D1%8F_%D0%BF%D0%B5%D1%80%D0%B5%D0%BC%D0%B5%D0%BD%D0%BD%D1%8B%D1%85_%D1%82%D0%B8%D0%BF%D0%B0) над ними). В системах типов второго порядка нет непосредственной необходимости в поддержке [примитивных типов](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B8%D0%BC%D0%B8%D1%82%D0%B8%D0%B2%D0%BD%D1%8B%D0%B9_%D1%82%D0%B8%D0%BF), так как они могут быть выражены посредством более развитых средств.[[15]](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%BB%D0%B8%D0%BC%D0%BE%D1%80%D1%84%D0%B8%D0%B7%D0%BC_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)#cite_note-_84c9e6bfe2806753-15)

Классическим примером полиморфного типа служит [*список*](https://ru.wikipedia.org/wiki/%D0%A1%D0%BF%D0%B8%D1%81%D0%BE%D0%BA_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5))*элементов произвольного типа*, для которого во многих языках, типизируемых по [Хиндли — Милнеру](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0_%D1%82%D0%B8%D0%BF%D0%BE%D0%B2_%D0%A5%D0%B8%D0%BD%D0%B4%D0%BB%D0%B8_%E2%80%94_%D0%9C%D0%B8%D0%BB%D0%BD%D0%B5%D1%80%D0%B0" \o "Система типов Хиндли — Милнера) (большинство [статически типизируемых](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D0%B0%D1%82%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B0%D1%8F_%D1%82%D0%B8%D0%BF%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F)[функциональных языков программирования](https://ru.wikipedia.org/wiki/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%BE%D0%BD%D0%B0%D0%BB%D1%8C%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)), предоставляется [синтаксический сахар](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D0%BD%D1%82%D0%B0%D0%BA%D1%81%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D1%81%D0%B0%D1%85%D0%B0%D1%80).

[Ad hoc](https://ru.wikipedia.org/wiki/Ad_hoc) полиморфизм (в русской литературе чаще всего переводится как «специальный полиморфизм» или «специализированный полиморфизм», хотя оба варианта не всегда верны) поддерживается во многих языках посредством [перегрузки функций и методов](https://ru.wikipedia.org/wiki/%D0%9F%D0%B5%D1%80%D0%B5%D0%B3%D1%80%D1%83%D0%B7%D0%BA%D0%B0_%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%B9), а в [слабо типизированных](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D0%B8_%D1%81%D0%BB%D0%B0%D0%B1%D0%B0%D1%8F_%D1%82%D0%B8%D0%BF%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F) — также посредством [приведения типов](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B8%D0%B2%D0%B5%D0%B4%D0%B5%D0%BD%D0%B8%D0%B5_%D1%82%D0%B8%D0%BF%D0%BE%D0%B2).

* 1. Полиморфные типы

**Переменная типа**- математическая переменная, которая может принимать значение из множества типовданных

**Полиморфный тип** - тип, операции которого могут также быть применены к ценностям некоторого другого типа или типов.

* 1. Ленивые и энергичные вычисления

При энергичной модели (называемой ещё «жадной» или «строгой») выражение, являющееся аргументом функции, будет вычислено ещё до того, как попадёт в тело функции. При ленивой же модели всё наоборот: выражение, являющееся аргументом функции, передаётся в функцию прямо так, без вычисления. Модель вычисления не влияет на результат этого вычисления, но она влияет на путь к этому результату. Однако, бывают ситуации, когда программа, написанная на языке с энергичной моделью вычислений, выдает ошибку, когда та же программа на языке с ленивыми все же выполнится.

Так, рассмотрим схематично некоторый код, в котором есть функция от двух переменных, возвращающая X/Y, если X>0 и X, если X<0. Если вызвать такую функцию с аргументами (-2, 0), но с разными моделями вычислений, то в языке с энергичной моделью программа выдаст сообщение об ошибке. Программа же с ленивыми вычислениями выдаст -2.

Ленивая модель вычислений позволяет нам работать с бесконечными структурами данных. Хотя, надо сказать, что не совсем бесконечными. Будет выдано столько элементов этой бесконечной структуры, сколько будет затребовано. Также на примере 7-го задания было показано, что поток, он же бесконечный список, может участвовать в формировании самого себя без каких-либо проблем (для решения этой же программы на языке с энергичной моделью вычислений потребовалось куда больше лишних действий).

* 1. Механизм вывода типов

В языке **Haskell** реализован ***механизм вывода типов***.

***Выводом типов*** называется ***автоматическое*** полное или частичное вычисление типов некоторых выражений программы, основанное лишь на её синтаксической структуре и производимое ***статически***, т.е. без необходимости запуска программы ***(достоинство)***.

Алгоритм и сама возможность вывода типов целиком зависят от используемой системы типов.

Исходная информация для алгоритма вывода типов - сама программа, способ присваивания типов литералам, типы некоторых функций и значений, а также правила типизации синтаксических конструкций (типы встроенных арифметических операций, типы операций доступа к структурам данных, взятые из определений этих структур, и т.п.).

Правила типизации синтаксических конструкций и литералов обычно фиксированы в рамках одного алгоритма вывода типов.

Наиболее известный, простой и широко применяющийся в алгоритм вывода типов - это ***алгоритм Хиндли-Милнера***(используемый, в частности, в типизированном  **λ**-исчислении).

Алгоритм, называемый **System F**, является основой систем типов большинства современных функциональных языков программирования, таких как **OCaml** и **Haskell**.

Таким образом, в большинстве случаев программисту не нужно явно указывать тип выражений, т.к. этот тип будет автоматически вычислен на основе контекста вычислительного процесса. ***(достоинство)***

Однако ***культура программирования на языке*** **Haskell** подразумевает явное указание типа используемых функций.

Механизм вывода типов основан на возможности автоматически полностью или частично выводить тип выражения, полученного при помощи вычисления некоторого выражения. Так как этот процесс систематически производится во время трансляции программы, транслятор часто может вывести тип переменной или функции без явного указания типов этих объектов. Во многих случаях можно опускать явные декларации типов — это можно делать для достаточно простых объектов, либо для языков с простым синтаксисом. Например, в языке Haskell реализован достаточно мощный механизм вывода типов, а сам синтаксис языка достаточно простой, поэтому указание типов функций в этом языке программирования не требуется. Программист может указать тип функции явно для того, чтобы ограничить её использование только для конкретных типов данных, либо для более структурированного оформления исходного кода.

Для того, чтобы получить информацию для корректного вывода типа выражения в условиях отсутствия явной декларации типа этого выражения, транслятор либо собирает такую информацию из явных деклараций типов подвыражений (переменных, функций), входящих в изучаемое выражение, либо использует неявную информацию о типах атомарных значений. Такой алгоритм не всегда помогает определить тип выражения, особенно в случаях использования функций высших порядков и параметрического полиморфизма достаточно сложной природы ***(недостаток)***. Поэтому в сложных случаях, когда есть необходимость избежать неоднозначностей, рекомендуется явно указывать тип выражений.

* 1. Модули

1) Модуль -  функционально законченный фрагмент программы, оформленный в виде отдельного файла с исходным кодом или поименованной непрерывной её части, предназначенный для использования в других программах.

2) Достоинствами модульного программирования является следующее:

· большую программу могут писать одновременно несколько программистов, что позволяет раньше закончить задачу;

· можно создавать библиотеки наиболее употребительных модулей;

· упрощается процедура загрузки в оперативную память большой программы, требующей сегментации;

· появляется много естественных контрольных точек для отладки проекта;

· проще проектировать и в дальнейшем модифицировать программы.

Например, в лабораторной работе №3 мы реализовали эту работу как модуль Lab3, внутри которого нами описывались функции. Далее при загрузе файла с тестами мы подключали модуль Lab3 (import Lab3), и применяли к содежащимся в нем функциям тесты.Также при создании стратегии к игре "Uno" необходимо было создать свой модуль и описать в нем стратегию, при необходимости, подобные модули удобно использовать при запуске игры с несколькими стратегиями.

* 1. Абстракции

1) Что понимается под термином "Абстракция данных"?

Абстракция данных является важным понятием, потому что она позволяет **создавать типы данных, не предос­тавляемые в языке программирования**. Еще одним преимуществом абстракции данных является возможность создавать *готовые элементы программ —* части про­граммного обеспечения, которые могут использоваться снова и снова в различ­ных программах.

Методика абстракции применяется и к данным. Каждый тип данных со­стоит из набора значений (области) и набора операций над этими значениями. Абстракция данных применяется, когда требует­ся использовать тип данных, не встроенный в язык программирования. Новый тип можно определить как *абстрактный тип данных (АТД),* сосредоточив внимание только на его логических свойствах, не заботясь о деталях их реализации.

Например АТД, Время:

ТИП

-Время

ОБЛАСТЬ

-Каждое значение типа Время – время в часах, минутах и секундах.

ОПЕРАЦИИ

-Установить время

-Вывести время на экран

-Увеличить время на 1 секунду

-Сравнить два значения времени на равенство

-Сбросить время

Спецификация АТД определяет для пользователя **абстрактные значения данных и абстрактные операции**. Конечно, в итоге АТД должен быть реализован в виде кода программы. Для реализации АТД, программист должен сделать две вещи

1. Выбрать конкретное *представление абстрактных данных,* используя уже су­ществующие типы данных.
2. Представить каждую из дозволенных операций в виде инструкций программы.

Чтобы реализовать АТД Время, для представления данных нужно исполь­зовать три переменные типа int - по одной для часов, минут и секунд. Либо мы можем использовать для представления три строки (три массива типа char) или даже массив тина int из трех элементов. Спецификация АТД не ограничивает нас каким-то определенным представлением данных. Пока мы удовлетворяем спецификации, мы вольны выбирать любое представление и связанные с ним алгоритмы. Наш выбор может основываться на эффективности по времени (ско­рости, с которой выполняется тот или иной алгоритм), экономии памяти или на простоте и читабельности алгоритмов.

* 1. Контейнеры

**Конте́йнер** в [программировании](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) — тип, позволяющий [инкапсулировать](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D0%BA%D0%B0%D0%BF%D1%81%D1%83%D0%BB%D1%8F%D1%86%D0%B8%D1%8F_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) в себе [объекты](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) других типов. Контейнеры, в отличие от коллекций, реализуют конкретную [структуру данных](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85).

Среди «широких масс» программистов наиболее известны контейнеры, построенные на основе [шаблонов](https://ru.wikipedia.org/wiki/%D0%A8%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD%D1%8B_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F), однако существуют и реализации в виде библиотек (наиболее широко известна библиотека [GLib](https://ru.wikipedia.org/wiki/GLib" \o "GLib)). Кроме того, применяются и узкоспециализированные решения. Примерами контейнеров в C++ являются контейнеры из стандартной библиотеки ([STL](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D0%B0%D0%BD%D0%B4%D0%B0%D1%80%D1%82%D0%BD%D0%B0%D1%8F_%D0%B1%D0%B8%D0%B1%D0%BB%D0%B8%D0%BE%D1%82%D0%B5%D0%BA%D0%B0_%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD%D0%BE%D0%B2)) — map, vector и др. В контейнерах часто встречается реализация алгоритмов для них.

Как следует из названия, контейнер — это объект, который может содержать в себе другие объекты. Существует несколько разных типов контейнеров. Например, класс vector определяет динамический массив, deque создает двунаправленную очередь, а list представляет связный список. Эти контейнеры называются последовательными контейнерами (sequence containers), потому что в терминологии STL последовательность — это линейный список. STL также определяет ассоциативные контейнеры (associative containers), которые обеспечивают эффективное извлечение значений на основе ключей. Таким образом, ассоциативные контейнеры хранят пары “ключ/значение”. Примером может служить map. Этот контейнер хранит пары “ключ/значение”, в которых каждый ключ является уникальным. Это облегчает извлечение значения по заданному ключу.

Контейнер, в отличие от [коллекции](https://dic.academic.ru/dic.nsf/ruwiki/120935), в общем случае, обычно не допускает явного задания числа элементов иобычно не поддерживает ветвистой структуры. Впрочем, это сильно зависит от реализации, посколькумногие реализации (особенно ориентированные на долговременное хранение) позволяют задавать размерыпри создании контейнера.  
Создание и использование последовательного контейнера предусматривает выполнение перечисленных ниже шагов. 1. Создать экземпляр нужного контейнера. В этом рецепте используется vector, но вместо него можно подставить любой последовательный контейнер. 2. Добавить элементы в контейнер вызовом insert(). 3. Получить количество элементов в контейнере вызовом size(). 4. Определить, пуст ли контейнер (т.е. факт отсутствия в нем элементов), вызовом empty(). 5. Удалить элементы из контейнера вызовом erase(). 6. Удалить все элементы из контейнера вызовом clear(). 7. Получить итератор, указывающий на начало последовательности вызовом begin(). Получить итератор, указывающий на позицию, находящуюся за концом последовательности, вызовом end(). 8. Для обратимых последовательных контейнеров получить обратный итератор, указывающий на конец последовательности, вызовом rbegin(). Получить обратный итератор, указывающий на позицию, предшествующую началу последовательности, вызовом rend(). 9. Выполнить цикл по элементам контейнера посредством итератора. 10. Обменять содержимое одного контейнера с содержимым другого посредством swap(). 11. Определить, когда один контейнер эквивалентен, меньше или больше другого.

* 1. Карринг
* Каррировать функцию - это преобразовать функцию, принимающую n аргументов в функцию, которая принимает один аргумент и возвращает каррированую функцию, которая принимает n — 1 аргументов.
* Для функции двух аргументов f: (A x B) -> C оператор каррирования Lambda выполняет преобразование Lambda (f): A -> (B -> C)  — берёт аргумент типа A и возвращает функцию типа B -> C. С интуитивной точки зрения, каррирование функции позволяет фиксировать её некоторый аргумент, возвращая функцию от остальных аргументов. Таким образом, Lambda — функция типа ((A x B) -> C) -> (A -> (B -> C)).

13. Замыкание

**Замыкание -**функция первого класса, в теле которой присутствуют ссылки на переменные, объявленные вне тела этой функции в окружающем коде и не являющиеся её параметрами.

Замыкание — это особый вид функции. Она определена в теле другой функции и создаётся каждый раз во время её выполнения. По сути, замыкание - функция, находящаяся целиком в теле другой функции. При этом вложенная внутренняя функция содержит ссылки на локальные переменные внешней функции. Каждый раз при выполнении внешней функции происходит создание нового экземпляра внутренней функции, с новыми ссылками на переменные внешней функции.

В случае замыкания ссылки на переменные внешней функции действительны внутри вложенной функции до тех пор, пока работает вложенная функция, даже если внешняя функция закончила работу, и переменные вышли из области видимости.

Замыкание связывает код функции с её лексическим окружением (местом, в котором она определена в коде).

14. Мемоизация

**Мемоизация**- один из способов оптимизации программы, увеличивающий скорость выполнение компьютерной программы. Заключается он в предотвращении повторных вычислений. Т.е. сохраняются результаты выполнения функций.

Алгоритм работы данного способа следующий:

* если функция не вызывалась ранее, то она вызывается, результат сохраняется
* если вызывалась - используется то, что получилось ранее

15. Динамическое окружение

Динамическое окружение для переменной — это когда связь, установленная в более внешнем вызове, остается в силе для всех вложенных контекстов, возникающих во время вычисления (при условии, что эта переменная снова не связывается). В различных участках программы одна и та же переменная может принимать значения разных типов. <прим. проверяющего: не из той оперы. Слово "Динамический" у нас применяется к нескольким понятиям и здесь речь не о том, что требуется>

Формирование динамического окружения впервые происходит во время выполнения программы и может измениться во время выполнения программы.

Окружение представляет собой набор переменных, к которым имеют доступ все исполняемые команды.

* val a = 25         [(a, 25)]
* val b = 17         [(b, 17), (a, 25)]
* val c = 7 + a     [(c, 32), (b, 17), (a, 25)]
* val a = 36         [(a,36), (c, 32), (b, 17), (a, 25)]
* val c = let          [(let 1, [(a,36), (c, 32), (b, 17), (a, 25)], (a, 17), (b, 36), (a, 36), (c, 32), (b, 17), (a,25)]
* val a = 17
* val b = 36
* in
* a + b
* end                  после let: [(a,36), (c, 32), (b, 17), (a, 25)]
* после операции связывания: [(c, 53), (a,36), (c, 32), (b, 17), (a, 25)]
* val c = c + 8 + b+ a                                 c, 93)[(, (c, 53), (a,36), (c, 32), (b, 17), (a, 25)]

16. Шаблоны

1) Шаблон - это один из вариантов определения функции, т.е. предположение о том, в какой виде аргументы могут подаваться в функцию. Для примера, обратимся к данному коду:

fun is\_nil (nil) = true

    | is\_nil (\_::\_) = false

2) is\_nil определяет структуру списков: функция определяется с помощью двух предположений - одно предположение для nil, а другое для hd::tl. В определении функции шаблоны (предположения) отделяются друг от друга вертикаьной чертой. В общем случаи, если тип аргумента имеет более одного конструктора, то определение функции должно содержать по одному предположению на каждый конструктор. Это гарантирует, что функция может принять любой аргумент данного типа.

Также определение функции с помощью разбора случаев применимо и для рекурсивных функций. Предположим мы хотим определить функцию append, которая получается в качестве аргументов два списка и возвращает список, полученный путем второго списка в конец первого:

- fun append (nil, lst) = lst

     | append (hd :: tl, lst) = hd :: append (tl, lst)

Один случай для пустого списка другой для непустого, добавление lst к пустому списку результатом яляется lst, в случаи непустого списка мы должны добавить lst к tl, и результат соединить в спискок  с hd.

3)

Чтобы избавиться от необходимости указывать имена, которые в дальнейшем все равно не будут использоваться, SML позволяет писать вместо них универсальный образец — «джокер» (обозначаемый знаком \_ подчеркивание), который может быть сопоставлен с любым значением без привязки к нему идентификатора.

 - val \_ :: tl = lst;

> val tl = ["and", "behold"] : string list

- val (\_, x) = (5, 6.7);

> val x = 6.7 : real

в Huskell иногда бывает удобно ввести «промежуточные» переменные в образце. Например, нам может понадобиться привязать к паре (ll, rr) идентификатор l. Это выполняется с помощью многоуровневых образцов. Многоуровневый образец получается путем приписывания образца к переменной внутри другого образца, как в следующем примере:

- val (l as (ll, lr), r) = x;   <прмеч. проверяющего: Пример на SML>

 > val l = ("fоо", true) : string \* bool

    val ll = "fоо" : string

    val lr = true : bool

    val r = 17 : int

Любая переменная может входить в образец только один раз. В частности, нельзя задать образец вроде (x, x) — который должен был бы быть сопоставим только с симметричными парами.

17. Объект первого класса

Объектами первого класса - это элементы, которые могут быть переданы как параметр, возвращены из функции, присвоены переменной.  
Объект первого класса:

Может быть сохранен в переменной или структурах данных;

Может быть передан в функцию как аргумент;

Может быть возвращен из функции как результат;

Может быть создан во время выполнения программы;

Внутренне самоопознаваем (независим от именования).

18. Модели вычислений

Стратегия вычисления (англ. evaluation strategy) — правила семантики языка программирования, определяющие, когда следует вычислять аргументы функции (метода, операции, отношения), и какие значения следует передавать.

Аппликативный порядок:

Аппликативный порядок вычислений (англ. applicativeorder), также «вычисления слева направо, изнутри наружу», (leftmostinnermost)[2][3] <ССЫЛКИ ЕСТЬ, А СПИСОК ЛИТЕРАТУРЫ НЕ ПРИЛОЖЕН. НЕПОРЯДОК>, означает стратегию вычислений, при которой обход снизу вверх по AST  <?????> вычисляет аргументы слева направо в редуцируемых выражениях.

В отличие от вызова по значению, аппликативный порядок вычислений максимально редуцирует термы в теле функции до её применения.

Для рассмотрения примера вычислений аппликативным порядком определим несколько функций:

square(x) = x \* x

sum\_of\_squares(x, y) = square(x) + square(y)

f(x) = sum\_of\_squares(x + 1, x \* 2)

При вычислении значения f(5) получим следующий набор подстановок:

f(5) = sum\_of\_squares(5 + 1, 5 \* 2) = square(6) + square(10) = ((6 \* 6) + (10 \* 10)) = 36 + 100 = 136

Нормальный порядок:

Нормальным порядком вычислений (англ. Normal order; также «вычислениями слева направо, снаружи внутрь», leftmost outermost) называют стратегию вычислений, при которой охватывающее выражение полностью редуцируется, применяя функции до вычисления аргументов.

В отличие от нормального порядка, стратегия «вызов-по-имени» не вычисляет аргументы и выражения внутри функций, которые не вызываются.

Например, значение f(5) для функции f, определенной ранее, при вычислении нормальным порядком даст следующий набор подстановок:

f(5) = sum-of-squares (5 + 1, 5 \* 2) = square(5 + 1) + square(5 \* 2) = ((5 + 1) \* (5 + 1)) + ((5 \* 2) \* (5 \* 2)) = (6 \* 6) + (10 \* 10) = 36 + 100 = 136

19. Рекурсивный процесс

 Рекурсия — это свойство объекта подражать самому себе. Объект является рекурсивным если его части выглядят также как весь объект. Рекурсия в программировании – это пошаговое разбиение задачи на подзадачи, подобные исходной.

 Рекурсия может быть прямой, когда функция вызывает сама себя (А->А), и косвенной, когда функция А вызывает функцию В, которая в свою очередь вызывает функцию А (А->В->А). При каждом вызове любой подпрограммы происходит помещение в системный стек данных этой программы, а именно: точки возврата, значений формальных параметров и локальных констант и переменных, то глубина рекурсии не может быть бесконечной, поскольку в конце концов этот системный стек переполнится.

 Если процедура или функция вызывает себя сама, это называют прямой рекурсией. Но может встретиться ситуация, когда подпрограмма обращается к себе опосредованно, путем вызова другой подпрограммы, в которой содержится обращение к первой. В этом случае имеем дело с косвенной, или взаимной, рекурсией.

 Рекурсивный процесс постоянно говорит «я это запомню и потом посчитаю» на каждом шаге рекурсии. «Потом» наступает в самом конце. а) Когда рекурсивный процесс считает факториал 6, то ему нужно запомнить 5 чисел чтобы посчитать их в самом конце, когда уже никуда не деться и рекурсивно двигаться вниз больше нельзя. б) Когда мы находимся в очередном вызове функции, то где-то снаружи этого вызова в памяти хранятся эти запомненные числа.

 Итеративный процесс постоянно говорит «я сейчас посчитаю все что можно и продолжу» на каждом шаге рекурсии. Ему не нужно ничего запоминать вне вызова, он всегда считает все в первый возможный момент, и каждый шаг рекурсии может существовать в изоляции от прошлых, потому что вся информация передается из шага в шаг. а) Когда итеративный процесс считает факториал 6, то ему не нужно запоминать числа. Нужно лишь считать и передавать результат дальше, в новый вызов. б) Когда мы находимся в очередном вызове функции, снаружи этого вызова в памяти ничего не нужно запоминать.

 Хвостовая рекурсия — специальный случай рекурсии, при котором рекурсивный вызов функцией самой себя является её последней операцией. Когда происходит вызов функции, компьютер должен запомнить место, из которого функция была вызвана (адрес возврата), чтобы после её окончания вернуться и продолжить выполнение программы. Обычно адрес возврата сохраняется в стеке. Иногда последнее действие функции после завершения всех других операций, это просто вызов функции, возможно самой себя, и возвращение результата. В этом случае нет необходимости запоминать адрес возврата, вновь вызываемая функция будет возвращать результат непосредственно к месту вызова первоначальной функции.

   Пример рекурсивной функции: возведение числа x в степень n с помощью умножения (рекурсия по аргументу):

   step  ::  Double -> Integer -> Double

   step x 0 = 1

   step x n = step x (n-1) \* x

   Пример хвостовой функции: вычисления длины заданного списка

     length\_a :: [a] -> Int

     length\_a l = lngt l 0

     lngt :: [a] -> Int-> Int

     lngt [] n = n

     lngt (x:xs) n = lngt xs (n  + 1)

20. Затенение

Затенение - это переопределение переменной с тем же именем.  
Смысл затенения: из нескольких одноименных переменных всегда имеется в виду более локальная переменная, то есть та, чья зона видимости меньше.

Механизм затенения: из нескольких одноименных переменных всегда имеется в виду более локальная переменная, то есть та, чья зона видимости меньше. То есть переменные, локальные в процедуре, имеют предпочтение перед переменными, локальными в модуле, а те - перед глобальными переменными. В этом есть глубокий смысл. Программист, объявляющий переменные в своей процедуре, может не заботиться о том, что где-то в модуле есть переменные с тем же именем. А программист, объявляющий переменные в своем модуле, может не заботиться о том, что переменные с тем же именем есть где-то в проекте.

Можно скрыть переменную, которую вы хотите скрыть двумя способами:

Затенения посредством области. Его можно скрыть путем ее переопределения внутри подобласти, которая содержит переменную, которую требуется скрыть.

Затенения посредством наследования. Путем ее переопределения в производном классе.

21. Лексическое и динамическое связывание

 Связывание — процесс установления некоторой связи, например между атрибутом и объектом или между операцией и символом. Момент установления связи называется временем связывания (binding time). Связывание может происходить во время разработки или реализации языка, а также в ходе компиляции, загрузки или выполнения программы. Например, символ звездочки (\*) обычно связывается с операцией умножения в процессе создания языка, а тип данных INT/INTEGER с некоторым диапазоном возможных значений целых чисел во время реализации языка.

 Статическое связывание типов может осуществляться с помощью некоторой формы явного или неявного объявления. Явное объявление — это оператор программы, перечисляющий имена переменных и устанавливающий определенный тип для них. Примером является оператор объявления переменной в разделе var на языке Pascal. Неявное объявление — это способ связывания переменных с типами в соответствии

с принятыми по умолчанию соглашениями, без использования специальных операторов. В этом случае первое появление имени переменной в программе является её неявным объявлением. Неявные объявления могут значительно снизить надежность программы и привести к появлению трудноуловимых ошибок, поскольку соответствующие переменные получают типы по умолчанию и способны иметь неожиданные атрибуты.

 При динамическом связывании в операторе объявления тип не указывается. Переменная связывается с типом при присвоении ей значения посредством оператора присваивания. Основное преимущество динамического вида связывания заключается в том, что оно обеспечивает значительную гибкость программирования, по сравнению со статическим. Однако динамическое связывание имеет два существенных недостатка. Во-первых, присвоение переменной одного типа значения другого типа не будет считаться ошибкой с точки зрения системы реализации языка. Поэтому данный тип связывания может послужить причиной появления трудноуловимых ошибок. Другой недостаток динамического связывания состоит в снижении эффективности выполнения программы, так как требуется дополнительное время на проверку и преобразование типов и перераспределение памяти, а также требуются дополнительные ресурсы для хранения в памяти сведений о текущем типе каждой переменной.

22. Параметры функции

Функция - это часть программы, оформленная в виде отдельной синтаксической конструкции и снабженная именем. «Вызов» функции ( подпрограммы), т.е. выполнение действий, заданных в подпрограмме в виде операторов, может быть произведен в некоторой точке программы посредством указания имени этой подпрограммы. Кроме задания последовательности действий, любая подпрограмма может содержать описание некоторой совокупности локальных объектов — констант, типов, переменных и т.д. Эти объекты предназначены для организации действий внутри подпрограммы и имеют смысл (т.е. доступны или видимы) только внутри данной подпрограммы. Параметры подпрограмм используются для обмена значениями между вызываемой и вызывающей частями программы. Описываемые в заголовке объявления подпрограммы параметры называются формальными, а те, которые подставляются на их место при вызове, — фактическими.

 — Параметры-значения (или передача параметров по значению). Параметры-значения — это локальные переменные подпрограмм, стартовые значения которых задаются при вызове подпрограмм из внешних блоков (им присваиваются значения соответствующих фактических параметров). Параметры-значения, описанные в заголовке, могут изменять свои значения наряду с прочими переменными, но эти изменения будут строго локальными и никак не отразятся на значениях фактических параметров.

 — Параметры-переменные (или передача параметров по ссылке). Параметры-переменные предоставляют принципиально иной механизм передачи параметров, состоящий в том, что в подпрограмму передается не значение фактического параметра, а ссылка на ячейку (или ячейки) памяти где хранится значение этого фактического параметра. При этом все действия с формальным параметром-переменной (внутри подпрограммы) на самом деле являются действиями над фактическим параметром.

 — Параметры-константы — аналогичны параметрам-значениям с той лишь разницей, что они не могут изменять свои значения в теле подпрограммы. Попытка изменить значение параметра-константы будет немедленно замечена компилятором.

23. Макросы

Макросы - это текстовая замена до начала работы самой программы.  
Рассмотрим описание макроса на примере факториала  
(defmacro fact (n)  
 (list 'if  
        (list ' < n 2)  
         1  
        (list '\* n  
              (list 'fact  
                    (list '1 - n)))))  
Описанный макрос будет производить текстовую замену и вычислять значение факториала, то есть:  
(if (< 5 2) 1  
  (\* 5 (fact (1 - 5))))  
Для проверки на что произойдёт замена используется вызов команды (macroexpand-1 '(fact 5))

У макросов существуют так называемые протечки, то есть когда макрос может работать некорректно.  
Основные варианты протечки:

**1. При определение макросов используем собственные имена**  
    Для того, чтобы избежать этого используем функцию gensym - функция с побочным эффектом, которая выдает нам имя идентификатора.   
**2. Когда определяем макрос с несколькими аргументами**  
    Например:  
(defmacro myIf (e1 e2 e3)  
  `(let ((p1, e2)  
          (p2, e3)  
          (p3, e1))  
   (cond (p3 p1)  
             (T p2))))  
Данный макрос, будет иметь протечку, так как вызывается от трех аргументов.  
**3. Вычисление аргументов лишнее число раз**

24. Статическая и динамическая типизация

Статическая типизация - это когда переменная, параметр подпрограммы, возвращаемое значение функции связывается с типом в момент объявления и тип не может быть изменён позже (переменная или параметр будут принимать, а функция — возвращать значения только этого типа). Примеры статически типизированных языков — Ада, С++, C#, D, Java, ML, Паскаль, Nim.

Динамическая типизация - это когда переменная связывается с типом в момент присваивания значения, а не в момент объявления переменной. Таким образом, в различных участках программы одна и та же переменная может принимать значения разных типов. Примеры языков с динамической типизацией — Smalltalk, Python, Objective-C, Ruby, PHP, Perl, JavaScript, Lisp, xBase, Erlang, Visual Basic.

Статические и динамические языки построены на фундаментально разных идеях о корректности программ. В динамическом языке "a" + 1 это корректная программа: код будет запущен и появится ошибка в среде исполнения. Однако, в большинстве статически типизированных языков выражение "a" + 1 — это не программа: она не будет скомпилирована и не будет запущена. Это некорректный код, так же, как набор случайных символов !&%^@\*&%^@\* — это некорректный код. Это дополнительное понятие о корректности и некорректности не имеет эквивалента в динамических языках.

25. Абстрактные типы данных

Абстрактный тип данных — это математическая модель для типов данных, где тип данных определяется поведением с точки зрения пользователя данных, а именно в терминах возможных значений, возможных операций над данными этого типа и поведения этих операций.

Формально АТД может быть определен как множество объектов, определяемое списком компонентов (операций, применимых к этим объектам, и их свойств). Вся внутренняя структура такого типа спрятана от разработчика программного обеспечения — в этом и заключается суть абстракции. Абстрактный тип данных определяет набор функций, независимых от конкретной реализации типа, для оперирования его значениями. Конкретные реализации АТД называются структурами данных. В программировании абстрактные типы данных обычно представляются в виде интерфейсов, которые скрывают соответствующие реализации типов. Программисты работают с абстрактными типами данных исключительно через их интерфейсы, поскольку реализация может в будущем измениться. Такой подход соответствует принципу инкапсуляции в объектно-ориентированном программировании. Сильной стороной этой методики является именно сокрытие реализации.

Раз вовне опубликован только интерфейс, то пока структура данных поддерживает этот интерфейс, все программы, работающие с заданной структурой абстрактным типом данных, будут продолжать работать. Разработчики структур данных стараются, не меняя внешнего интерфейса и семантики функций, постепенно дорабатывать реализации, улучшая алгоритмы по скорости, надежности и используемой памяти.

26. Отражения

Для каждого объекта можно узнать характеристики данного объекта с помощью отражений. С помощью них можно узнать, к примеру:

  .class – к какому классу принадлежит объект;

 .superclass – какой класс является родительским для класса объекта;

 .methods – список всех методов, которые можно применить к объекту;

 .nil? – является ли объект nil'ом

 .is\_a? – принадлежит ли объект к указанному классу или его подклассу (принимает класс (например, x.is\_a? Integer) и возвращает значение true, если объект является экземпляром этого класса или его подкласса);

 .instance\_of? – принадлежит ли объект указанному классу (принимает класс и возвращает true, только если класс, для которого вызывается метод непосредственно является экземпляром указанного класса).

27. Преимущества статической/динамической типизаци

**Преимущества статистической типизации**

* Статическая типизация даёт самый простой [машинный код](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%88%D0%B8%D0%BD%D0%BD%D1%8B%D0%B9_%D0%BA%D0%BE%D0%B4). Поэтому она удобна для языков, дающих [исполняемые файлы](https://ru.wikipedia.org/wiki/%D0%98%D1%81%D0%BF%D0%BE%D0%BB%D0%BD%D1%8F%D0%B5%D0%BC%D1%8B%D0%B9_%D1%84%D0%B0%D0%B9%D0%BB) [ОС](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0) или [JIT-компилируемые](https://ru.wikipedia.org/wiki/JIT-%D0%BA%D0%BE%D0%BC%D0%BF%D0%B8%D0%BB%D1%8F%D1%86%D0%B8%D1%8F) [промежуточные коды](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%BC%D0%B5%D0%B6%D1%83%D1%82%D0%BE%D1%87%D0%BD%D1%8B%D0%B9_%D0%BA%D0%BE%D0%B4).
* Многие ошибки исключаются уже на стадии компиляции.
* Поэтому статическая типизация хороша для написания сложного, но быстрого кода.
* В [интегрированной среде разработки](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D0%B3%D1%80%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D1%80%D0%B5%D0%B4%D0%B0_%D1%80%D0%B0%D0%B7%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%BA%D0%B8) осуществимо более релевантное [автодополнение](https://ru.wikipedia.org/wiki/%D0%90%D0%B2%D1%82%D0%BE%D0%B4%D0%BE%D0%BF%D0%BE%D0%BB%D0%BD%D0%B5%D0%BD%D0%B8%D0%B5" \o "Автодополнение), особенно если типизация — сильная статическая: множество вариантов можно отбросить как не подходящие по типу.
* Чем больше и сложнее проект, тем большее преимущество дает статическая типизация, и наоборот.

**Недостатки статистической типизации**

* Языки с недостаточно проработанной математической базой оказываются довольно многословными: каждый раз надо указывать, какой тип будет иметь переменная. В некоторых языках есть автоматическое [выведение типа](https://ru.wikipedia.org/wiki/%D0%92%D1%8B%D0%B2%D0%B5%D0%B4%D0%B5%D0%BD%D0%B8%D0%B5_%D1%82%D0%B8%D0%BF%D0%B0), однако оно может привести к трудноуловимым ошибкам. Нивелируется функциями IDE вроде quick fix.
  + Сказанное не верно для языков семейства [ML](https://ru.wikipedia.org/wiki/ML), основанных на т. н. «главной типизации» ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) principal typing scheme), которая одновременно автоматически [выводит](https://ru.wikipedia.org/wiki/%D0%92%D1%8B%D0%B2%D0%B5%D0%B4%D0%B5%D0%BD%D0%B8%D0%B5_%D1%82%D0%B8%D0%BF%D0%B0) наиболее общий тип всякого выражения и строго проверяет согласование типов зависимостей. Это придаёт языку выразительность динамически типизируемых, но обеспечивает лучшее быстродействие и [типобезопасность](https://ru.wikipedia.org/wiki/%D0%A2%D0%B8%D0%BF%D0%BE%D0%B1%D0%B5%D0%B7%D0%BE%D0%BF%D0%B0%D1%81%D0%BD%D0%BE%D1%81%D1%82%D1%8C" \o "Типобезопасность)[[3]](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D0%B0%D1%82%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B0%D1%8F_%D1%82%D0%B8%D0%BF%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F#cite_note-_418d7e4ab389749e-3).
* Тяжело работать с данными из внешних источников (например, в [реляционных СУБД](https://ru.wikipedia.org/wiki/%D0%A0%D0%B5%D0%BB%D1%8F%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D0%A1%D0%A3%D0%91%D0%94) / десериализация данных).

**Преимущества динамической типизации**

* Облегчается работа прикладного программиста с [СУБД](https://ru.wikipedia.org/wiki/%D0%A1%D0%A3%D0%91%D0%94), которые принципиально возвращают информацию в «динамически типизированном» виде. Поэтому динамические языки ценны, например, для программирования [веб-служб](https://ru.wikipedia.org/wiki/%D0%92%D0%B5%D0%B1-%D1%81%D0%BB%D1%83%D0%B6%D0%B1%D0%B0).
* Иногда требуется работать с данными переменного типа. Например, функция [поиска подстроки](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%B8%D1%81%D0%BA_%D0%BF%D0%BE%D0%B4%D1%81%D1%82%D1%80%D0%BE%D0%BA%D0%B8) возвращает позицию найденного символа (число) или маркер «не найдено». В [PHP](https://ru.wikipedia.org/wiki/PHP) этот маркер — булевое false. В языках со статической типизацией это особая константа (0 в [Паскале](https://ru.wikipedia.org/wiki/%D0%9F%D0%B0%D1%81%D0%BA%D0%B0%D0%BB%D1%8C_(%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)), std::string::npos в [C++](https://ru.wikipedia.org/wiki/C%2B%2B)).

**Недостатки динамический типизации**

Статическая типизация позволяет уже при компиляции заметить простые ошибки «по недосмотру». Для динамической типизации требуется как минимум выполнить данный участок кода.

* В [объектно-ориентированных](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) языках не действует, либо действует с ограничениями, [автодополнение](https://ru.wikipedia.org/wiki/%D0%90%D0%B2%D1%82%D0%BE%D0%B4%D0%BE%D0%BF%D0%BE%D0%BB%D0%BD%D0%B5%D0%BD%D0%B8%D0%B5" \o "Автодополнение): трудно или невозможно понять, к какому типу относится переменная, и вывести набор её полей и методов.
* Развитая статическая система типов (такая как [Хиндли-Милнер](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0_%D1%82%D0%B8%D0%BF%D0%BE%D0%B2_%D0%A5%D0%B8%D0%BD%D0%B4%D0%BB%D0%B8_%E2%80%94_%D0%9C%D0%B8%D0%BB%D0%BD%D0%B5%D1%80%D0%B0" \o "Система типов Хиндли — Милнера)) играет значительную роль в самодокументировании программы; динамическая типизация по определению не проявляет этого свойства, что затрудняет разработку структурно сложных программ.
* Снижение производительности из-за трат процессорного времени на динамическую проверку типа, и излишние расходы памяти на переменные, которые могут хранить «что угодно». К тому же большинство языков с динамической типизацией интерпретируемые, а не компилируемые.

**Какой вид типизации лучше?**

Споры о том, что динамическая типизация лучше, чем строгая, не прекращаются и по сей день. Обычно они возникают у узкоспециализированных программистов. Безусловно, веб-разработчики повседневно используют все преимущества динамической типизации для создания качественного кода и итогового программного продукта. В то же время системные программисты, которые разрабатывают сложнейшие алгоритмы на низкоуровневых языках программирования, обычно не нуждаются в таких возможностях, поэтому им вполне хватает статической типизации. Бывают, конечно, исключения из правил. Например, полностью реализована динамическая типизация в Python. Поэтому определять лидерство той или иной технологии, нужно исходя только из входных параметров. Для разработки легких и гибких фреймворков лучше подойдет динамическая типизация, в то время как для создания массивной и сложной архитектуры лучше использовать строгую типизацию.

28. Классы и объекты. Составные части.

В Ruby все типы являются объектами, даже сами классы. Каждый класс до конца выполнения программы остаётся открытым, а это значит, что в любой тип можно добавить собственные методы (или изменить поведение существующих). Каждый класс можно определять постепенно, в нескольких частях программы. Добавленные методы становятся доступны немедленно, в том числе для уже созданных экземпляров типа. Стоит помнить, что методы в Ruby — на самом деле «сообщения», и у каждого метода есть «приёмник», то есть объект, которому сообщение отправлено. Метод по умолчанию ищет другие методы в экземпляре класса, поскольку приёмником для него является self.

Операция расширения класса (добавление нового метода к существующему) по сути не отличается от создания нового класса.

У объектов в Ruby есть методы класса и методы экземпляра. В нашем примере consonants — это именно метод экземпляра. При создании нового класса или изменении существующего создать метод класса можно, начав его имя с имени класса или с self и точки.

Если перед именем какой-либо константы стоит @, то она является переменной экземпляра класса или переменной объекта.   
Например: @dog, @a, @abc, @length.

Областью видимости этих переменных является объект. Список переменных объекта можно получить используя метод #instance\_variables. Получить значение переменной экземпляра класса можно используя метод #instance\_variable\_get.

Переменные класса — это переменные, имена которых начинаются с @@.   
Например: @@heigth, @@tail.   
Переменные классов отличаются от переменных экземпляра класса тем, что будучи объявлены в любом месте, их областью видимости будет класс и все его экземпляры. То есть в отличие от переменных объекта самого класса, к переменным класса можно обратиться напрямую.

Константы — это «переменные», чье значение не изменяется, а имя начинается с заглавной литеры.  
Например: Fieldsize, All, Array

В Ruby возможно переопределять значение константы, однако мы получим предупреждение.

28. Двойная диспетчеризация

Двойная диспетчеризация - механизм, который отправляет вызов функции различным конкретным функциям в зависимости от времени выполнения двух объектов, участвующих в вызове.  
Чтобы осуществить двойную диспетчеризацию нужно:  
1. Использовать однократную диспетчеризацию дважды  
2. Основной метод немедленно вызывает другой метод по своему первому параметру, передавая self(себя) как аргумент.

3. Второй вызов будет косвенно знать класс self параметра.  
4. Он также будет знать класс первого параметра основного метода из-за однократной диспетчеризации.

29. Инкапсуляция

Инкапсуляция в Ruby достигается за счет различных степеней ограничения доступа к методам объекта, только с помощью вызова которых другой объект может взаимодействовать с данным. Значение по умолчанию – public (общий), что означает, что любой объект любого класса может вызвать такой метод. Описание метода как private (частный, приватный), как в случае с переменными экземпляров, позволяет только методам самого объекта вызывать такой метод. Описание метода как protected (защищенный) означает, что такой метод может вызываться любым объектом, который является экземпляром того же класса или любого подкласса данного класса. Существуют различные способы, чтобы задать видимость метода. Самый простой – в рамках определения класса, можно указать ключевое слово public, private или protected перед описанием соответствующего блока методов. Предполагается неявное указание public перед описанием первого метода в классе. В Ruby есть синтаксическая особенность у private-методов: если метод m объявлен как private, то его можно вызывать только как m(args). Вызовы как x.m(args) или даже self.m(args) не допускаются.

Любой объектно-ориентированный язык предоставляет механизм инкапсуляции. В общепринятом смысле это означает, во-первых, что атрибуты и методы объекта ассоциированы именно с этим объектом, а во-вторых, что область видимости атрибутов и методов по умолчанию ограничена самим объектом (применение принципа сокрытия информации). Инкапсуляция говорит о том, что нам достаточно знать лишь интерфейс класса, чтобы им пользоваться. Мы не должны знать ничего о внутренней структуре класса, чтобы с ним взаимодействовать. В Ruby мы тоже можем легко узнать интерфейс класса — посмотреть на его публичные методы. Затем найти метод, который нам подходит и посмотреть, какие параметры передать, чтобы получить желаемый результат. Ruby — язык с динамической типизацией, поэтому все, что мы можем узнать о методе с первого взгляда — это его имя и количество параметров, которые он принимает. Следует отметить, что именование публичных методов и параметров играет важную роль, так как служит своеобразной документацией метода. Не следует забывать, что класс должен иметь как можно меньше публичных методов. Это позволит упростить работу с классом и защитить данные от нежелательных изменений со стороны. Все «служебные» методы должны быть приватными. Public, private и protected методы в Ruby. Public методы можно запускать откуда угодно, так же можно обратиться через «.». Private методы используются только объектами класса, нельзя обратиться через «.». Protected методы используются объектами класса и представителями всех дочерних классов, из экземпляра класса или его наследника можно обратиться через «.».

30. Способы передачи параметров

Когда одна функция вызывает другую, обычный метод сообщения между ними состоит в использовании глобальных переменных, возвращаемых значения и параметров вызываемой функции.

В языках программирования имеется два основных способа передачи параметров подпрограмме. Первый из них – *передача по значению*. При его применении в формальный параметр подпрограммы копируется значение фактического параметра (аргумента). В таком случае изменения формального параметра на фактический аргумент не влияют.

Вторым способом передачи параметров подпрограмме является *передача по ссылке*. При его применении в формальный параметр копируется адрес фактического аргумента. Это значит, что, в отличие от передачи по значению, изменения значения формального параметра приводят к точно таким же изменениям значения фактического аргумента.

31. Бэктрекинг

Бэктрекинг – это перебор с возвратами, который организуется автоматически при попытке унифицировать цель. Автоматический перебор с возвратами – существенное преимущество Пролога, так как освобождает программиста от необходимости явной организации такого перебора. Процесс доказательства в Прологе основан на переборе в глубину. Сначала выбирается первая по порядку альтернатива, которая может быть использована для доказательства цели. Если происходит возврат в точку выбора (в случае неуспеха доказательства), то выбирается следующая альтернатива, и так далее. Процесс перебора вариантов будет продолжаться до тех пор, пока не будет доказан запрос. Кроме того, пользователь может инициировать процесс возврата после получения решения, если он захочет получить другой вариант решения.

Предикат отсечения, обозначаемый символом «!», позволяет уменьшить пространство поиска и сократить перебор, отбрасывая неиспользованные альтернативы. Выполнение отсечения приводит к следующим результатам. Отбрасываются все предложения для согласуемой цели, расположенные после правила, содержащего отсечение. Отбрасываются все альтернативные решения конъюнкции целей, расположенных в предложении левее отсечения. С другой стороны, отсечение не влияет на цели, расположенные правее его. В случае возврата они могут порождать альтернативные решения. Если доказательство этих целей не будет успешным, происходит возврат к последнему выбору, сделанному перед выбором предложения, содержащего отсечение. Пусть программа содержит следующее предложение: a :- b, c, !, d, e. После выполнения предиката ! (в случае успешного доказательства целей b и c), в случае неуспеха доказательства целей d и e альтернативы для c, b, a не рассматриваются, и доказательство цели a завершается неуспехом. Предикат отсечения можно сравнить с барьером, через который не может перейти процесс возврата.

Можно выделить три основных случая использования отсечения:

1. подтверждение правильности выбора правила;

2. прекращение доказательства цели;

3. завершение последовательности порождения и проверки вариантов.

32. Полиморфизм

Полиморфизм (поли — много, морфе — форма) буквально означает "способность принимать разные формы и обличия". Существует наследственный полиморфизм и интерфейсный полиморфизм.   
Реализация интерфейсного полиморфизма у языка Ruby в виде модулей (которые позволяют создавать примеси в классах).  
Наследственный полиморфизм используется в классе-потомке и позволяет переопределить унаследованные атрибуты и методы класса-родителя. В примере мы переопределили, метод initialize дополнив его атрибутом @height\_range, а также мы использовали выражение super() для того чтобы выполнить родительский одноименный метод, в данном случае это конструктор класса . Если бы мы не вызывали родительский одноименный метод, то конструктор родительского класса бы не выполнился. Иными словами super() нужен для дополнения "устаревших" версий методов класса-родителя, при  переопределении оных классом потомком.

33. Парадигмы программирования

Императивная парадигма программирования описывает процесс вычислений посредством описания управляющей логики программы, т.е. в виде последовательности отдельных команд, которые должен выполнить компьютер. Каждая команда является инструкцией, изменяющей состояние программы. Программа, написанная в императивном стиле, похожа на набор приказов, выражаемых повелительным наклонением в естественных языках.

Декларативная парадигма программирования определяет процесс вычислений посредством описания логики самого вычисления, а не управляющей логики программы.

Декларативное программирование является противоположностью императивного программирования; первое описывает, что необходимо сделать, а второе — как именно это сделать. Фактически, декларативное программирование часто определяется как любой стиль программирования, не являющийся императивным, т.к. последнему легче дать определение.

Функциональное программирование — раздел дискретной математики и парадигма программирования, в которой процесс вычисления трактуется как вычисление значений функций в математическом понимании (то есть тех, чей единственный результат работы заключается в возвращаемом значении, или другими словами, вычисление которых не имеет побочного эффекта). Противопоставляется парадигме императивного программирования, в которой исполнителю программы предписывается последовательность выполняемых действий, в то время, как в функциональном программировании способ решения задачи описывается при помощи зависимости функций друг от друга (в том числе возможны рекурсивные зависимости), но без указания последовательности шагов. После выполнения программы не остаётся каких-либо следов от вызова программы.

Одной из близких парадигм программирования является логическое программирование, в котором программа представляет собой множество пар (логическое условие, новые факты). В логическом программировании, также как и в функциональном программировании, программист остается в неведении о методах, применяемых при вычислении, и последовательности исполнения элементарных действий. Большая часть ответственности за эффективность вычислений в логическом и функциональном программировании перекладывается на «плечи» транслятора используемого языка программирования.

Функциональное и логическое программирование являют собой части т. н. «декларативного программирования», т. е. такого стиля программирования, при использовании которого в программах описывается способ решения поставленной задачи, а не предписываются шаги для получения результата.

Параллельное программирование представляет программу в виде набора сообщающихся процессов, которые могут выполняться параллельно. Такие программы могут выполняться как на одном процессоре (чередуя выполнение шагов каждого процесса), так и на нескольких.

Объектно-ориентированная парадигма (ООП) представляет программу как набор объектов и их взаимодействий. Основными понятиями ООП являются следующие:

объект — элементарная сущность, описываемая определенными свойствами (хранящимися в виде атрибутов объекта) и поведением (реализованным в виде методов);

класс описывает структуру свойств и поведения одного типа объектов. Каждый объект программы является экземпляром некоторого класса;

классы могут наследовать атрибуты и методы их родительских классов, в то же время добавляя свои собственные. Иерархия классов позволяет моделировать сущности решаемой задачи на нескольких уровнях детализации и в дальнейшем использовать класс, отвечающий уровню детализации, необходимому для решения конкретной подзадачи.

инкапсуляция подразумевает, что некоторые детали реализации класса скрыты от взаимодействующих с ним объектов. У каждого класса есть интерфейс, описывающий взаимодействие объектов этого класса с прочими объектами, и реализация, описывающая то, как это взаимодействие отражается на объекте этого класса.

34. Моделирование отложенных вычислений

Отложенные вычисления, ленивые вычисления или нестрогие вычисления (англ. lazy evaluation) — концепция в некоторых языках программирования, согласно которой вычисления следует откладывать до тех пор, пока не понадобится их результат.

Отложенные вычисления позволяют сократить общий объём вычислений за счёт тех вычислений, результаты которых не будут использованы. Программист может просто описывать зависимости функций друг от друга и не следить за тем, чтобы не осуществлялось «лишних вычислений».

Отложенные вычисления естественным образом легли на функциональную парадигму программирования, поскольку функциональные языки программирования, реализующие отложенные вычисления, зарекомендовали себя как инструменты, удобные для прототипирования и быстрой разработки программного обеспечения, а также для проектирования электронно-вычислительных устройств.

Языки программирования, реализующие отложенные вычисления

Haskell — язык программирования, в котором наиболее полно реализованы отложенные вычисления. Не имеет оператора присваивания значения переменной, а только операцию определения функциональной зависимости. Имеет списки отложенных вычислений, позволяющие программистам оперировать бесконечными последовательностями.

Mathematica и Maxima допускают отложенные вычисления с помощью оператора определения «:=».

35. Система типов

**Полная / надежная**

Система типизации называется надежной (sound), если она отвергает все программы, которые могут выполнить недопустимые действия X. Система типизации называется полной (complete), если она не отвергает ни одной программы, которая не выполнит X. К примеру, выражение if true then 5 else "abc" в языке с надежной системой типизации (таком, как Standart ML) не пройдет компиляцию, т.к. ветки then и else имеют разный тип. При этом, если бы система типизации была полной, то выражение выполнилось бы, т.к. система распознала бы, что противоречие никогда не допускается. Невозможно создать такую систему типизации, которая одновременно была бы надежной, полной и заканчивала компиляцию (т.е. отвергала все неверные программы, принимала все работающие программы и не была бы бесконечной). Предпочитают отказаться от чего-то одного. Чаще всего оставляют sound (надежную) систему типизации, а не complete (полную).

**Сильная / слабая типизация**(также иногда говорят *строгая / нестрогая*).

Сильная типизация выделяется тем, что язык не позволяет смешивать в выражениях различные типы и не выполняет автоматические неявные преобразования, например нельзя вычесть из строки множество.

Языки со слабой типизацией выполняют множество неявных преобразований автоматически, даже если может произойти потеря точности или преобразование неоднозначно.

Примеры: Сильная: Java, Python, Haskell, Lisp; Слабая: C, JavaScript, Visual Basic, PHP.

**Явная / неявная типизация.**

Явно-типизированные языки отличаются тем, что тип новых переменных, функций, их аргументов нужно задавать явно.

Соответственно языки с неявной типизацией перекладывают эту задачу на компилятор / интерпретатор.

Примеры: Явная: C++, D, C# Неявная: PHP, Lua, JavaScript

Также нужно заметить, что все эти категории пересекаются, например язык C имеет статическую слабую явную типизацию, а язык Python — динамическую сильную неявную.

36. Унификация

Сопоставление (унификация) является наиболее важной операцией в языке Prolog. Сопоставление выполняет сравнение двух термов на равенство, при этом неконкретизированные переменные получают значения, при которых термы становятся идентичными. Выполнение сопоставления может производиться либо явно, в теле правила с помощью встроенного предиката X=Y, либо неявно, при сопоставлении цели с фактом или головой правила. Сопоставление реализует основные операции обработки данных в логическом программировании: — однократное присваивание — передача параметров — создание структурных объектов — доступ к полям структурных объектов с возможностью одновременного чтения/записи Сопоставление выполняется согласно следующим правилам: — Не конкретизированная переменная сопоставима с любым объектом и этот объект становится значением переменной (конкретизацией) — Числа и атомы сопоставимы только с идентичными числами и атомами — Структуры сопоставимы только, если они имеют одинаковый функтор, одинаковое число компонентов, и соответствующие компоненты сопоставимы друг с другом — Не конкретизированные переменные сопоставимы друг с другом, при этом они становятся сцепленными. Если одна из них получит конкретное значение, то такое же значение получит и другая переменная

37. Наследование

Наследование – это ключевое понятие объектно-ориентированного программирования.

Каждый класс неявно наследуется от Object. Сам Object наследуется от BasicObject.

Наследование обозначается символом <.

class A < B (Класс А является наследником класса B)

Иерархию наследования можно представлять в виде дерева. Высота дерева может быть любым, это значит, что

Если для дочернего класса вызывается метод родительского класса, то метод  вызывается от класса, который встретится раньше. То есть мы последовательно ищем метод в родительских классах, пока не встретим нужный метод.

38. Утиная типизация

**Латентная**, **неявная** или **утиная**[**типизация**](https://dic.academic.ru/dic.nsf/ruwiki/4896) (калька с [англ.](https://dic.academic.ru/dic.nsf/ruwiki/6161) *Duck typing*) — вид динамической типизации, применяемой в некоторых языках программирования ([Perl](https://dic.academic.ru/dic.nsf/ruwiki/7468), [Smalltalk](https://dic.academic.ru/dic.nsf/ruwiki/6036), [Objective-C](https://dic.academic.ru/dic.nsf/ruwiki/41166), [Python](https://dic.academic.ru/dic.nsf/ruwiki/1447), [Ruby](https://dic.academic.ru/dic.nsf/ruwiki/5721), [Groovy](https://dic.academic.ru/dic.nsf/ruwiki/128876), [ColdFusion](https://dic.academic.ru/dic.nsf/ruwiki/10929), [Boo](https://dic.academic.ru/dic.nsf/ruwiki/151307), [Java](https://dic.academic.ru/dic.nsf/ruwiki/1315), [C#](https://dic.academic.ru/dic.nsf/ruwiki/1112)), когда границы использования объекта определяются его *текущим набором методов и свойств*, в

противоположность наследованию от определённого класса. То есть считается, что объект реализует интерфейс, если он содержит все методы этого интерфейса,

независимо от связей в иерархии наследования и принадлежности к какому-либо конкретному классу.

Название термина пошло от английского *«duck test»* («[тест на утку](https://dic.academic.ru/dic.nsf/ruwiki/694345)»), который в оригинале звучит как:

*«If it looks like a duck, swims like a duck and quacks like a duck, then it probably is a duck».*

(*«Если это выглядит как утка, плавает как утка и крякает как утка, то, вероятно, это утка».*).

### Назначение

Утиная типизация решает такие проблемы иерархической типизации, как:

* невозможность явно указать (путем наследования) на совместимость интерфейса со всеми настоящими и
* будущими интерфейсами, с которыми он идейно совместим;
* экспоненциальное увеличение числа связей в иерархии типов при хотя бы частичной попытке это сделать.

Другим близким подходом является структурные подтипы в [OCaml](https://dic.academic.ru/dic.nsf/ruwiki/5251), где типы объектов совместимы, если

совместимы сигнатуры их методов, независимо от объявленного наследования, причём всё это проверяется вовремя компиляции программы.

39. Идентификаторы

 Объектами первого класса в контексте конкретного языка программирования называются элементы, которые могут быть переданы как параметр, возвращены из функции или присвоены переменной. Самые простые типы данных — это числа и строки.

 Идентификатором называется последовательность цифр, букв и специальных символов. При этом первой стоит буква или специальный символ. Для получения идентификаторов можно использовать строчные или прописные буквы латинского алфавита. Специальным символом может служить символ подчеркивания (). Два идентификатора, для получения которых применяются совпадающие строчные и прописные буквы, считают различными. К примеру: abc, ABC, A328B, a328b. Компилятор допускает всякое количество символов в идентификаторе, но значим только первый 31 символ. Идентификатор образуется на этапе объявления переменной, функции, структуры и т. п. После этого его можно применять в последующих операторах разрабатываемой программы. Важно отметить некоторые особенности при выборе идентификатора. Во-первых, идентификатор и ключевое слово не должны совпадать. Также не должно быть совпадения с зарезервированными словами и названиями функций библиотеки компилятора языка СИ.

 Во-вторых, важно обратить особое внимание на применение символа подчеркивания () первым символом идентификатора, так как идентификаторы выстраиваются так, что, с одной стороны, могут совпадать с именами системных функций и (или) переменных, но при этом при применении таких идентификаторов программы могут стать непереносимыми, т. е. их нельзя применять на компьютерах других типов.

 В-третьих, на идентификаторы, применяемые для определения внешних переменных, должны быть наложены ограничения, которые формируются используемым редактором связей. Кроме того, использование различных версий редактора связей или различных редакторов определяет различные требования на имена внешних переменных.