Экзамен

1) Понятие о данных.

Данные - представление фактов, понятий, инструкций в форме приемлемой для обмена, интерпретации или обработки человеком или с помощью автоматических средств.

2) Понятия программы, алгоритма.

Компьютерная программа - алгоритм, записанный на некотором языке программирования.

Язык программирования - формальный язык, предназначенный для записи компьютерных программ.

Алгоритм - конечная совокупность точно заданных правил решения произвольного класса задач или набор инструкций, описывающий порядок действий исполнителя для решения некоторых задач.

Свойства алгоритма:

1. Дискретность - наличие структуры, разбитие на отдельные команды, понятия, действия.
2. Детерминированность - для одного и того же набора данных всегда один и тот же результат.
3. Понятность - элементы алгоритма должны быть понятны исполнителю.
4. Завершаемость - алгоритм имеет конечное количество шагов.
5. Массовость - один алгоритм применим к некоторому классу задач.
6. Результативность - алгоритм должен выдавать результат.

3) Понятия типа данных, системы типов языка программирования, типизации.

Типы данных - множество значений, множество операций над ними и способ хранения в памяти компьютера (машинное представление).

Система типов - совокупность правил в языках программирования, назначающих свойства, именуемые типами, различным конструкциям, составляющим программу — переменные, выражения, функции и модули.

Система типов по Пирсу - разрешимый синтаксический метод доказательства отсутствия определённых поведений программы путём классификации конструкции в соответствии с видами вычисляемых значений.

Классификация систем типов:

1. **Наличие системы типов**: есть/нет
   Нет: ASM, FORTH, B.
   Есть: все остальные языки
2. Типизация: **статическая/динамическая**
   Статическая: C, C++, Java, Haskell, Rust, Go.
   Динамическая: Scheme, JavaScript, Python
3. Типизация: **явная/неявная**
   Явная (явно записывается): C, C++, Java.
   Неявная (можно не записывать): C++(auto), Go(когда тип не указан), Rust, Haskell
4. Типизация: **сильная/слабая**
   Сильная (неявные преобразования типов запрещены): Scheme, Python, Haskell.
   Слабая (неявные преобразования допустимы): JavaScript, C, Perl, PHP (`'1000' * 5 -> 5000`)

Первая классификация типов:

1. **Простые** - неделимые порции данных: число, символ, литера (Scheme: `52`, `'a` C: `int`, `"a"`).
2. **Составные** - содержащие значения других типов: cons-ячейки, список, вектор, строка (Scheme: `(1, 2, 3)` C: `{1, 2, 3}`).

Вторая классификация типов:

1. **Встроенные типы данных** - уже заранее есть в языке.
2. **Пользовательский** - их определяет пользователь.
   Пользовательские типы данных часто представляют как списки, первым элементом которых является символ с именем типа, а остальные — хранимые значения.

4) Важнейший парадигмы программирования и их отличительные черты.

Парадигмы программирования - совокупность идей и понятий, определяющих стиль написания компьютерных программ (подход к программированию). Это способ концептуализации, определяющий организацию вычислений и структурирование работы, выполняемой компьютером.

Основные парадигмы программирования:

1. **Императивное программирование** - способ записи программ, в котором указывается последовательность действий, изменяющих состояние вычислительной среды (изменение памяти, ввод, вывод). Отличительная черта: деструктивное присваивание.
2. **Декларативное программирование** - способ записи программ, в котором описываются взаимосвязь между данными; описывается цель, а не последовательность шагов для её достижения.
3. **Метапрограммирование** - программа рассматривается как данные.

Основные подходы императивной парадигмы:

1. **Структурное программирование** - программа рассматривается как набор "фрагментов" кода (в т.ч. и вложенных друг в друга), имеющих один вход и один выход. Конструкции структурного ЯП: примитивный оператор (присваивание, вызов процедуры), ветвление, цикл.
2. **Процедурное программирование** - в рамках этого подхода программа рассматривается как набор отдельных подпрограмм, которые могут вызвать друг друга.
3. **Объектно-ориентированное программирование** - программа рассматривается как набор взаимодействующих объектов, объекты сочетают в себе состояние (данные) и поведение (связанные с объектом функции).

Основные подходы декларативной парадигмы:

1. **Функциональное программирование** - алгоритм описывается как набор функций в математическом смысле (функция - отображение входных данных на выходные). Единица декомпозиции программы - функция
   Чистые функции - детерминированные функции без побочного эффекта.
2. **Логическое программирование (Prolog)** - алгоритм описывает взаимосвязь между понятиями; выполнение программы сводится к выполнению запросов.

Основные подходы метапрограммирования парадигмы:

1. **Программы пишут программы** - макросы, генераторы кода, метапрограммирование шаблонов в C++
2. **Программы взаимодействуют с вычислительной средой** - рефлексия или интроспекция, программа анализирует свойства самой себя.

5) Каким образом в язык программирования с динамической типизацией можно ввести новый тип данных? Приведите

примеры.

Для этого достаточно придумать способ хранения данных нового типа и написать соответсвующие процедуры/методы для выполнения нужных операций.

Например, интервалы и интервальная арифметика (задача из лабораторной работы на Python) или многомерные вектора (в Scheme) или точка, представленная в виде списка из двух элементов, ее координат.

6) Понятие абстрактного типа данных. Примеры.

Абстрактный тип данных - множество значений и множество операций над ними, т.е. способ хранения не задан.

В ряде языков программирования (например, в Си) есть встроенные в язык средства для определения пользовательских типов данных. Например, в Си встроены различные числовые типы. Пользователь на их основе может создавать массивы, массивы массивов, структуры, объединения и т.д.

В языке Scheme нет языковых средств для определения новых типов данных. Вместо этого пользователь придумывает способ представления некоторого значения при помощи встроенных типов данных и описывает операции над ним в виде набора процедур (иногда, макросов).

Т.е. проектируем представление типа данных и набор операций. При этом не рекомендуется работать с типом данных в обход предоставленных операций.

Если мы задокументируем только набор операций, но не опишем представление, то мы создали абстрактный тип данных.

Примеры: стек, очередь, ассоциативный массив.

7) Функции (процедуры) высшего порядка в языках программирования высокого уровня.

Функция высшего порядка - функция принимающая в качестве аргументов другие функции или возвращающая другую функцию в качестве результата.

8) Способы организации повторяющихся вычислений в языках программирования высокого уровня.

Рекурсия, циклы, функции.

9) Мемоизация результатов вычислений.

Мемоизация — оптимизация, позволяющая избегать повторного вычисления функции, вызванной с теми же аргументами, как и в один из прошлых вызовов.

В общем случае можно мемоизировать чистые функции: детерминированные и без побочных эффектов. Обоснование:

- Нет смысла мемоизировать функцию (read port) — она на каждом вызове должна выдавать очередное значение из файла.
- Нет смысла мемоизировать функцию (display message), т.к. она должна выполнять побочный эффект.

```
(define memoized-factorial
  (let ((memo '()))
    (lambda (n)
      (let ((memoized (assq n memo)))
        (if memoized
            (cadr memoized)
            (let ((new-value
                   (if (< n 2) 1
                       (* (memoized-factorial ( n 1)) n))))
              (set! memo (cons (list n new-value) memo)) new-value))))))
```

10) Нестрогие и отложенные вычисления. Примеры. !!!

Строгие вычисления — аргументы функции полностью вычисляются до того, как эта функция вызывается. Вызовы процедур в Scheme всегда строгие. Строгую стратегию вычислений часто называют call-by-value. В случае **нестрогих вычислений** значения выражений могут вычисляться по необходимости, их вызов может быть отложен.

Примеры нестрогих вычислений:

- (if cond then else) — вычисляется либо then, либо else.
- (and …), (or …).
- В языке Си логические операции &&, || тоже не строгие.

В теории рассматривают две разновидности нестрогих вычислений:

- call-by-name, вызов по имени — нормальная редукция в лябмда-исчислении,
- call-by-need, вызов по необходимости — ленивые вычисления.

Отложенные (ленивые) вычисления - вычисления откладываются до тех пор, пока не понадобится их результат.

Отложенные вычисления можно организовать с помощью `delay` и `force` в Scheme и функций-генераторов и ключевого слова `yield` в Python.

В некотором смысле разновидность call-by-name — макроподстановка

11) Способы реализации языка программирования высокого уровня.

Способы реализации языка программирования: **интерпретация и компиляция**.

При **интерпретации** компьютер читает программу на языке программирования и выполняет инструкции, записанные в этой программе.

Преимущества: удобство разработки (нет промежуточной фазы компиляции), удобство отладки, переносимость.
Недостатки: низкое быстродействие, зависимость от интерпретатора.

При **компиляции** компьютер переводит программу с человекочитаемого языка на машинный язык. Транслятор — синоним компилятора.

Преимущества: высокое быстродействие, автономность готовых программ.
Недостаток: фаза компиляции, зависимость готовых программ от платформы.

Гибридный подход: компилятор формирует промежуточный более низкоуровневый код, который затем выполняется интерпретатором.

12) Компилятор и интерпретатор: определение, основные функциональные элементы.

Компьютер исполняет машинный код — последовательность примитивных операций: сложение двух ячеек памяти, пересылка значения из одной ячейки в другую, обращение к устройству, передача управления на другую инструкцию (безусловная или условная — в зависимости от результата предыдущей операции).

Стадии компиляции:

1. **Лексический анализ программы** — программа делится на «слова» — **лексемы,** некоторые небольшие структурные элементы: знаки операций, идентификаторы, литеральные константы (числа, строки, символы…). На стадии лексического анализа отбрасываются комментарии и символы пустого пространства (пробелы, табуляции, переводы строк). **Лексема** — подстрока исходной программы: `)`, `counter`, `007`. **Токен** — «обработанная» лексема, токен состоит из метки типа, позиции в исходном файле и атрибута (значения лексемы): `('CLOSE-BRACKET (1 1))`, `('IDENT (2 1) "counter")`, `('NUMBER (2 10) 7)`.

2. **Синтаксический анализ** — принимает последовательность токенов и строит из них синтаксическое дерево. Последовательность токенов плоская, выход синтаксического анализатора иерархичен — отражает структуру программы.
3. **Семантический анализ** — проверяет допустимость операций, правильность имён переменных, функций…
4. **Генерация промежуточного представления.** Преобразование синтаксического дерева в промежуточную форму, удобную для оптимизирующих преобразований (например, постфиксный код или SSA-форму).
5. **Оптимизации.** Эквивалентные преобразования программы, улучшающие временные характеристики.
6. **Генерация машинного кода.** Преобразование промежуточного представления в машинный код.
7. **Компоновка.** Программа может состоять из отдельно транслируемых файлов, в том числе библиотек — нужно из них собрать единую готовую программу.

Стадии 1–3 — стадии анализа (front end), стадии 4–7 — стадии синтеза (back end).

Стадии интерпретации.

1. **Лексический анализ.**
2. **Синтаксический анализ.**
3. **Семантический анализ.**
4. **Исполнение** (интерпретация) построенной программы.

13) Лексический анализатор: назначение, входные данные, выходные данные, принцип реализации.

Назначение: разбивает исходный текст на последовательность токенов, которые синтаксический анализ будет группировать в дерево. Либо, если исходный текст не соответствует грамматике — выдача сообщения (сообщений)об ошибке.

Входные данные: последовательность символов программы, записанной на исходном языке. (строка символов (или список символов))

Выходные данные: последовательность токенов. Можно сказать, что дерево разбора для грамматики лексем вырожденное — рекурсия есть только по правой ветке (cdr).

Распознавание лексем в контексте грамматики обычно производится путём их идентификации (или классификации) согласно идентификаторам (или классам) токенов, определяемых грамматикой языка. При этом любая последовательность символов входного потока (лексема), которая согласно грамматике не может быть

идентифицирована как токен языка, обычно рассматривается как специальный токен-ошибка.

Каждый токен можно представить в виде структуры, содержащей идентификатор токена (или идентификатор класса токена) и, если нужно, последовательность символов лексемы, выделенной из входного потока (строку, число и т. д.).

14) Синтаксический анализатор: назначение, входные данные, выходные данные.

Назначение: построение синтаксического дерева из списка токенов. Либо выдача сообщения об ошибке.

Входные данные: список токенов.

Выходные данные: дерево разбора (или синтаксическое дерево).

15) Формальная грамматика: терминальные символы, нетерминальные символы, правила, аксиома.

Формальная грамматика — набор правил, позволяющих породить строку, принадлежащую данному формальному языку. Формальная грамматика состоит из

- аксиомы,
- множества терминальных символов,
- множества нетерминальных символов,
- множества правил грамматики.

Терминальные символы — символы алфавита, из которых строятся строки данного формального языка.

Нетерминальные символы — символы, которые раскрываются согласно правилам грамматики.

Правила грамматики описывают, как в строке символов (терминальных и нетерминальных) раскрываются нетерминальные символы.

Аксиома грамматики — нетерминальный символ, выбранный в качестве стартового.

16) БНФ и синтаксические диаграммы.

Форма Бэкуса-Наура (БНФ) — способ описания грамматики, где правила имеют вид `<Нетерминал> ::= альтернатива | … | альтернатива` нетерминалы записываются в угловых скобках (<…>), терминальные символы записываются или сами собой (для

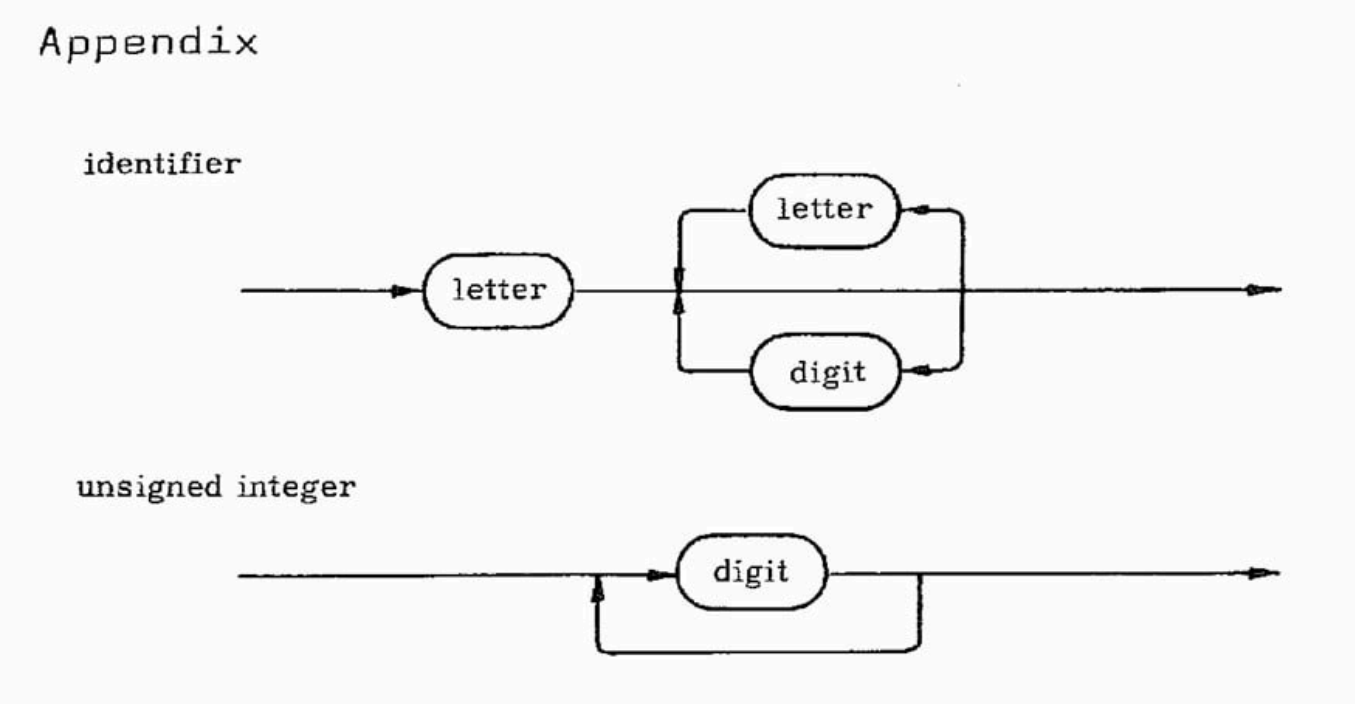
знаков операций, например), или словами БОЛЬШИМИ БУКВАМИ. Альтернативные варианты разделяются знаками |.

Пример:

```
<Выражение> ::= <Слагаемое> | <Выражение> + <Слагаемое>
<Слагаемое> ::= <Множитель> | <Слагаемое> * <Множитель>
<Множитель> ::= ЧИСЛО | ( <Выражение> )
```

Впервые она была использована при описании Алгола-60.

Синтаксические диаграммы — это графический способ описания грамматики языка. Впервые был использован Никлаусом Виртом при описании синтаксиса языка Паскаль в 1973 году.



17) *LL*(1)-грамматика: особенности и их использование.

Это такая грамматика, которая может быть использована для реализации LL(1) парсера, то есть нисходящего парсера. При этом 1 - число симовлов исходной последовательности, которые нужно рассмотреть в каждый момент анализа.

LL(k)-грамматики — грамматики, в которых мы можем определить правило для раскрытия нетерминала по первым `k` символам входной цепочки.

Дано: цепочка терминальных символов и нетерминальный символ. Требуется определить, по какому правилу нужно раскрыть нетерминальный символ, чтобы получить префикс этой цепочки. Для `LL(k)`-грамматик это можно сделать, зная первые k

символов.
Чаще всего рассматриваются `LL(1)-грамматики`, где раскрытие определяется по первому символу.

18) Принцип построения лексического анализатора.

Лексический анализатор должен определить границы лексем, которые в тексте исходной программы явно не указаны. Также он должен выполнить действия для сохранения информации об обнаруженной лексеме (или выдать сообщение об ошибке).

Причины использования лексического анализатора:

- применение лексического анализатора сокращает объем информации, обрабатываемой на этапе синтаксического разбора;
- некоторые задачи, требующие использования сложных вычислительных методов на этапе синтаксического анализа, могут быть решены более простыми методами на этапе лексического анализа (например, задача различения унарного минуса и бинарной операции вычитания, обозначаемых одним и тем же знаком "-");
- лексический анализатор отделяет сложный по конструкции синтаксический анализатор от работы непосредственно с текстом исходной программы, структура которого может варьироваться в зависимости от архитектуры вычислительной системы, где выполняется компиляция, — при такой конструкции компилятора для перехода на другую вычислительную систему достаточно только перестроить относительно простой лексический анализатор.

В основном лексические анализаторы выполняют исключение из текста исходной программы комментариев и незначащих символов (пробелов, символов табуляции и перевода строки), а также выделение лексем следующих типов: идентификаторов, строковых, символьных и числовых констант, ключевых (служебных) слов входного языка, знаков операций и разделителей.

Лексический анализатор имеет дело с такими объектами, как различного рода константы и идентификаторы (к последним относятся и ключевые слова)

19) Принцип построения нисходящего синтаксического анализатора, осуществляющего разбор методом рекурсивного спуска.

Метод рекурсивного спуска — способ написания синтаксических анализаторов для LL(1)-грамматик на алгоритмических языках программирования. Для каждого нетерминала грамматики записывается процедура, тело которой выводится из правил для данного нетерминала.

Написание синтаксического анализатора состоит из этапов:

1. Составление LL(1)-грамматики для данного языка
2. Формальное выведение парсера из правил грамматики. Парсер либо молча принимает строку, либо выводит сообщение об ошибке.
3. Наполнение парсера семантическими действиями — построение дерева разбора, выполнение проверок на корректность типов операций, возможно даже, вычисление результата в процессе разбора.

```
Expr    ::= Term Expr' .
Expr'   ::= AddOp Term Expr' | .
Term    ::= Factor Term' .
Term'   ::= MulOp Factor Term' | .
Factor  ::= Power Factor' .
Factor' ::= PowOp Power Factor' | .
Power   ::= value | "(" Expr ")" | unaryMinus Power.
```

20) Основные понятия объектно-ориентированного программирования.

ООП - парадигма программирования, в которой основными концепциями является понятие объектов.

Класс - абстрактный тип данных, объединяющий поля данных и методы их обработки.

Объект - воплощение (экземляр) какого-либо класса.

Наследование - это процесс, посредством которого один объект может наследовать свойства другого объекта и добавлять к ним черты, характерные для него.

Полиморфизм - возможность объектов с одинаковой спецификацией иметь различную реализацию.

Инкапсуляция - механизм языка программирования, позволяющий ограничить доступ одних компонентов программы к другим.

Членами класса являются атрибуты (свойства) и методы (возвращают или изменяют состояние объекта).

21) Модульное тестирование (юнит-тестирование), разработка через тестирование.

Разработка через тестирование — способ разработки программы, предполагающий написание модульных тестов (unit tests) до написания кода, который они проверяют.

Модульный тест — автоматизированный тест, проверяющий корректность работы небольшого фрагмента программы (процедуры, функции, класса и т.д.). Модульный тест обязательно должен быть самопроверяющимся, т.е. без контроля пользователя запускает тестируемую часть программы и проверяет, что результат соответствует ожидаемому.

Цикл разработки через тестирование:

1. Пишем тест для нереализованной функциональности. Этот тест при запуске проходить не должен.
2. Пишем функциональность, но ровно на столько, чтобы новый тест проходил. При этом все остальные тесты тоже должны проходить (не сломаться).
3. Рефакторинг — это эквивалентное преобразование программы, направленное на улучшение её внутренней структуры (повышение ясности программы, её расширяемости, эффективности). В процессе рефакторинга ни один из модульных тестов сломаться не должен.

Продолжительность одного цикла — около минуты.

22) Способы разбора и вычисления значения арифметического выражения, записанного в инфиксной нотации, с учетом приоритетов операций и скобок.

Существует как минимум два способа. Один из них - алгоритм сортировочной станции.

Также можно разбирать выражение с учетом приоритетности непосредственно во время синтаксического анализа (то есть во время работы парсера).

Лексический анализатор

Разрабатывается грамматика разбора арифметического выражения на токены и записываеся в БНФ перед главной процедурой лексера. Процедура принимает выражение в виде строки и возвращает последовательность токенов в виде списка. Лексемы исходной последовательности преобразованы:

- имена переменных и знаки операций — в символические константы Scheme,
- числа — в числовые константы Scheme соответствующего типа,
- скобки в — строки "(" и ")".

Синтаксический анализатор
Синтаксический анализатор должен строить дерево разбора согласно следующей

грамматике, учитывающей приоритет операторов:

```
Expr ::= Term Expr'
Expr' ::= AddOp Term Expr' | .
Term ::= Factor Term' .
Term' ::= MulOp Factor Term' | .
Factor ::= Power Factor' .
Factor' ::= Powop Power Factor' | .
Power ::= value | "(" Expr ")" | unaryMinus Power •
```

где терминалами являются value (число или переменная), круглые скобки и знаки операции.

Синтаксический анализатор принимает последовательность токенов в виде списка (результат работы tokenize) и возвращает дерево разбора, представленное в виде вложенных списков вида (операнд-1 знак-операции операнд-2) для бинарных операций и (-операнд) для унарного минуса.

23) Основные постулаты языков программирования семейства Lisp.

Постулаты:

1. Единство кода и данных.
2. Всё есть список.
3. Выражения являются списком, операция указывается в первом элементе.
4. Все выражения вычисляют значения.

24) Общая характеристика языка Scheme.

LISP (от LISt Processing) — язык программирования, созданный Джоном МакКарти в 1950-1960-е годы. Породил целое семейство языков со сходным синтаксисом и идеологией: Common Lisp, Scheme, Closure и т.д.

Scheme - это функциональный язык программирования, один из двух наиболее распространенных диалектов языка Lisp.

Scheme — язык семейства LISP, созданный Гаем Стилом и Джеральдом Сассманом в 1970-е годы. Отличается простотой и минималистичным дизайном.

25) Типизация и система типов языка Scheme.

Типизация: есть, динамическая, неявная, строгая

26) Способы определения процедуры в языке Scheme. Формальные и фактические аргументы, применение к аргументам, возвращаемое значение.

Определение процедуры

```
(define (add x y) (+ x y))
```

Вызов процедуры

```
(add 5 4)
; или
(apply add '(5 4))
```

apply применяет процедуру к списку аргументов.

- x, y - формальные аргументы
- 5, 4 - фактические аргументы

27) Простые типы языка Scheme и основные операции над ними.

Простые типы в Scheme:

- Числа
- Целые (например: 1, 4, -3, 0)
- Вещественные (например: 0.0, 3.5, 1.23E+10)
- Рациональные (например: 2/3, 5/2)

```
(number? 42)       =>  #t
(complex? 2+3i)    =>  #t
(rational? 2+3i)   =>  #f
(= 42 #f)          => ERROR!!!
(< 3 2)            =>  #f
(>= 4.5 3)         =>  #t
(+ 1 2 3)          =>  6
(- 5 2 1)           =>  2
(* 1 2 3)          =>  6
(/ 6 3)            =>  2
(/ 22 7)           =>  22/7
```

```
(expt 2 3)         =>  8
(max 1 3 4 2 3)    =>  4
(min 1 3 4 2 3)    =>  1
(abs 4)            =>  4
```

• Символы (например: `'a`)
Чтобы указать символ, нужно использовать символ цитирования (quote).

```
(symbol? 'xyz) =>  #t
(symbol? 42)   =>  #f
```

Символы в Scheme обычно не чувствительны к регистру

```
(equ? 'Calorie 'calorie) => #t
```

Мы можем использовать символ `xyz` как глобальнаую переменнаую при помощи определения: `(define xyz 9)` Это говорит о том, что переменная хранит в себе `9`, и, если мы посмотрим, что лежит в `xy`, то увидим:

```
xyz => 9
```

Мы можем использовать форму `set!`, чтобы изменить значение переменной:

```
(set! xyz #\c)
xyz => #\c
```

• Логический: Scheme использует специальные символы `#f` и `#t` для обозначения правды и лжи.

```
(boolean? #t)              =>  #t
(boolean? "Hello, World!") =>  #f
```

Процедура not отрицает свои аргументы.

```
(not #f)              =>  #t
(not #t)              =>  #f
(not "Hello, World!") =>  #f
```

• Characters (например: `#\a`)

```
(char? #\c)       =>  #t
(char? 1)         =>  #f
(char=? #\a #\a)  =>  #t
(char<? #\a #\b)  =>  #t
(char>=? #\a #\b) =>  #f
;ci == caseinsensitive
(charci=? #\a #\A) =>  #t
(chardowncase #\A) =>  #\a
(charupcase #\a)   =>  #\A
```

28) Составные типы языка Scheme и основные операции над ними.

Составные типы данных строятся путем соединения значений других типов данных.

- Строки – это последовательности, состоящие из characters:

```
(string #\h #\e #\l #\l #\o)     =>  "hello"
(string-ref "Hello; Hello!" 0)   =>  #\H
```

Новые строки могут быть созданы присоединением других строк:

```
(string-append "We"
               "are"
               "the"
               "champions") =>  "We are the champions"
```

Так же можно изменять значение какого-либо символа строки:

```
(string-set! hello 1 #\a)
hello =>  "Hallo"
```

- Векторы – это последовательности, как и строки, но их элементами могут быть любые типы данных, а не только символы. Элементами могут быть даже сами векторы, что является хорошим способом для получения многомерных векторов.

```
(vector 0 1 2 3 4) =>  #(0 1 2 3 4)
```

По аналогии с `make-string` , процедура `make-vector` создает вектор заданной длины:

```
(define v (make-vector 5))
```

Процедуры `vector-ref` и `vector-set!` обращаются к элементам и изменяют их. Предикат для проверки на принадлежность к типу вектора `vector?`.

- Точечные пары и списки
  Точечная пара получается путем соединения двух любых произвольных значений в упорядоченную пару. Первый элемент называется `car` , второй элемент называется `cdr` , а процедура объединения `cons`. Точечные пары нужна указывать самостоятельно, цитируя их (использовать символ `'`).

```
'(1 . #t) =>  (1 . #t)
(1 . #t)  =>  ERROR!!!
(define x (cons 1 #t))
(car x) =>  1
(cdr x) =>  #t
```

Элементы точечной пары могут быть заменены процедурой `set-car!` или `set-cdr!`:

```
(set-car! x 2)
(set-cdr! x #f)
x =>  (2 . #f)
```

Точечные пары могут содержать в себе другие точечные пары:

```
(define y (cons (cons 1 2) 3))
y =>  ((1 . 2) . 3)
```

`c...r` - сокращение для уровня вложенности "хвоста" пары. Например: `cadr`, `cddr`, и `cdddr` все действительны.

Но в Scheme есть процедура, которая называется `list`, котоая делает создание списков более удобным. Список принимает любое количество аргументов и возвращает список, содержащий их:

```
(list 1 2 3 4) =>  (1 2 3 4)
'(1 2 3 4)     =>  (1 2 3 4)
```

К элементу списка можно обратиться по индексу:

```
(define y (list 1 2 3 4))
(list-ref y 0)          =>  1
(list-ref y 3)          =>  4
(list-tail y 1)         =>  (2 3 4)
(list-tail y 3)         =>  (4)
```

Предикаты: `pair?`, `list?`, и `null?` проверяют аргументы на принадлежность к точечной паре, списку или пустому списку:

```
(pair? '(1 . 2)) =>  #t
(pair? '(1 2))   =>  #t
(pair? '())      =>  #f
(list? '())      =>  #t
(null? '())      =>  #t
(list? '(1 2))   =>  #t
(list? '(1 . 2)) =>  #f
(null? '(1 2))   =>  #f
```

- Преобразования типов данных

Схема предлагает множество процедур для преобразования между типами данных.

Characters могут быть преобразованы в целые числа с помощью `char->integer` , и целое число может быть преобразовано в characters: `integer->char` (целое число, соответствующее его ASCII-коду).

Строки могут быть конвертированы в соответствующий список characters. `(string->list "hello") -> (#\h #\e #\1 #\1 #\o)`

Также есть следующие процедуры перевода:

```
list->string
vector->list
list->vector

(number->string 16)                  =>  "16"
(string->number "16")                =>  16
(string->number "Am I a hot number?") =>  #f
```

29) Символьный тип в языке Scheme и его применение.

Слово «символ» в русском языке, применительно к типам в ЯП, двузначно: это и печатные знаки (из которых состоят строки), и некоторые имена (например, часть

компилятора — таблица символов (symbol table) хранит в себе свойства именованных сущностей — переменных, функций, типов и т.д.).

Литерный тип хранит в себе печатные знаки, т.е. знаки, которые вводятся с клавиатуры и выводятся на экран. Из литер состоят строки.

Символьный тип данных — это «зацитированное», «замороженное» имя: `'hello`

Операция цитирования — это особая форма (qoute …), которая принимает терм и «цитирует» его — все идентификаторы в нём становятся символами, остальные атомарные значения остаются как есть, выражения превращаются в списки.

30) Применение процедур (функций) высшего порядка для обработки списков на языке Scheme.

Функции высших порядков - это такие функции, которые могут принимать в качестве аргументов и возвращать другие функции.

Функции высших порядков позволяют использовать **карринг** - преобразование функции от пары аргументов в функцию, берущую свои аргументы по одному. Это преобразование получило свое название в честь Х. Карри.

```
(define (makevallist v0 dv vN)
  (define (makelist v)
    (if (= v vN)
    (cons v '())
    (cons v (makelist (+ v dv)))))
  (makelist v0))

(define Alist (makevallist 1 1 3))
(define xlist (makevallist 0.2 0.1 1.2))
(define (y A)
  (lambda (x) ( (* A x) (tan (* pi (/ x 4))))))
(define ylist (map (lambda (A)
              (map (y A) xlist))(начало  см. вопрос 9)*
            Alist))
(display (map (lambda (ls) (apply max ls)) ylist))
```

В примере применялись функции высшего порядка `map` и `apply`, они работают следующим образом:

- `map` применяет переданную ей в качестве аргумента функцию к каждому элементу списка, переданного вторым аргументом, и возвращает список значений-результатов применения, например:

```
(map abs '(-2 1 7 0 -5 4))  => '(2 1 7 0 5 4))
```

- `apply` применяет переданную ей в качестве первого аргумента функцию ко всем элементам списка, переданного вторым аргументом, как будто они (элементы списка) являются параметрами переданной функции, например функции:

```
(apply max '(-2 1 7 0 -5 4)) => 7
```

И в завершении вернемся к каррингу. Как видно из описания функции `map`, она может оперировать только функциями одной переменной. Вот тут-то к нам и приходят на помощь карринг и замыкания.

31) Лексические замыкания (на примере) в языке Scheme. Свободные и связанные переменные. Использование лексических замыканий для локальных определений (запись конструкций let и let* с помощью анонимных процедур).

```
(define (f x) (lambda (y) (+ x y)))
(define f1 (f 1))
(define f7 (f 7))

(f1 10) → 11
(f7 10) → 17
```

Замыкание — ситуация, когда из процедуры возвращается другая процедура, «запоминающая» локальные переменные окружающей процедуры. Здесь замыканием является `(lambda (y) (+ x y))`, т.к. она захватывает параметр `x` (который является локальной переменной) процедуры `f`.

Свободная переменная - переменная, которая встречается в теле функции, но не является её параметром и/или определена в месте, находящемся где-то за пределами функции.

Связанные переменные - переменные, которые либо являются параметрами данной функции, либо определены внутри этой функции.

Макросы `let` и `let*` через анонимные процедуры:

```
(let ((x 2) (y 3)) (+ x y)) <=> ((lambda (x y) (+ x y)) 2 3)
(let* ((x 2)
       (y (+ x 3))
```

```
        (z (* y 2)))
  (+ x y z)) <=>
((lambda (x)
   ((lambda (y)
      ((lambda (z) (+ x y z)) (* y 2)))
    (+ x 3)))
 2)
```

32) Лексические замыкания (на примере) в языке Scheme. Свободные и связанные переменные. Использование лексического замыкания для определения процедуры со статической переменной.

Статическая переменная сохраняет своё значение между вызовами процедуры, в которой она объявлена.

Пример использования лексических замыканий для определения процедуры со статической переменной:

```
(define counter
  (let ((n 0))
    (lambda ()
      (set! n (+ 1 n))
      n)))
(list (counter) (counter) (counter)) => (1 2 3)
```

33) Особенности логических операций в языке программирования Scheme.

Прежде всего стоит заметить, что в Scheme только значение `#f` является ложным. Все остальные стандартные значения в Scheme (включая пустые списки) считаются истинными.

Особенности процедуры `(and ...)` : Выражения вычисляются слева направо, и возвращается результат первого выражения значение которого ложно (`#f`), следующие за ним выражения не вычисляются. Если же все выражения истинны, то возвращается результат последнего выражения. Если выражений нет, то возвращается `#t` .

```
(and (= 2 2) (> 2 1)) ; ===>  #t
(and (= 2 2) (< 2 1)) ; ===>  #f
(and 1 2 'c '(f g))   ; ===>  (f g)
(and)                 ; ===>  #t
```

Особенности процедуры `(or ...)`: Выражения вычисляются слева направо, и возвращается результат первого выражения значение которого истинно, следующие за ним выражения не вычисляются. Если же все выражения ложны, то возвращается результат последнего выражения. Если выражений нет, то возвращается `#f`.

```
(or (= 2 2) (> 2 1))  ; ===>  #t
(or (= 2 2) (< 2 1))  ; ===>  #t
(or #f #f #f)         ; ===>  #f
(or (memq 'b '(a b c))
    (/ 3 0))          ; ===>  (b c)
```

34) Гигиенические макросы в языке Scheme.

Макрос — инструмент переписывания кода.
Процедура применяется к значениям во время выполнения и порождает новое значение.
Макрос применяется к фрагменту кода и порождает код до его выполнения.

Макросы Scheme инструмент, который позволяет расширять синтаксис языка, создавая новые конструкции. Определение макроса начинается с команды `define-syntax`.

```
(define-syntax macro
    (syntax-rules (<keywords>)
        ((<pattern>) <template>)
        ...
        ((<pattern) <template>)
    )
)
```

`<keywords>` — Ключевые слова, которые можно будет использовать в описании шаблона. *Например*, можно написать макрос для конструкции "(forech (item in items)..)", в данном случае ключевым словом будет "in", которое обязательно должно присутствовать.

`<pattern>` - Шаблон, описывающий, что на входе у макроса.

`<template>` — Шаблон, описывающий, во что должен быть трансформирован. В макросе многоточие "..." означает, что тело может содержать одну или более форм.

Макросы в Scheme **гигиенические**, т.е. о конфликте имён при их раскрытии беспокоиться не нужно.
Ключевые слова в макросе не являются метапеременными и трактуются буквально.

35) Продолжения в языке Scheme.

Продолжение (англ. continuation) представляет состояние программы в определённый момент, которое может быть сохранено и использовано для перехода в это состояние.

Продолжения содержат всю информацию, чтобы продолжить выполнения программы с определённой точки. Состояние глобальных переменных обычно не сохраняется, однако для функциональных языков это несущественно.

`call-with-current-continuation` (обычно сокращенно обозначается как `call/cc`) – функция одного аргумента, который мы будем называть получатель (receiver). Получатель также должен быть функцией одного аргумента, называемого продолжение.

`call/cc` формирует продолжение, определяя контекст выражения (call/cc receiver) и обрамляя его в функцию выхода escaper. Затем полученное продолжение передается в качестве аргумента получателю.

Если мы возьмём текущее продолжение и сохраним его где-нибудь, мы тем самым сохраним текущее состояние программы - заморозим её. Это похоже на режим гибернации ОС. В объекте продолжения хранится информация, необходимая для возобновления выполнения программы с той точки, когда был запрошен объект продолжения. Операционная система постоянно так делает с вашими программами, когда переключает контекст между потоками.

Разница лишь в том, что всё находится под контролем ОС. Если вы запросите объект продолжения (в Scheme это делается вызовом функции `call-with-current-continuation`), то вы получите объект с текущим продолжением - стеком. Вы можете сохранить этот объект в переменную (или даже на диск). Если вы решите "перезапустить" программу с этим продолжением, то состояние вашей программы "преобразуется" к состоянию на момент взятия объекта продолжения. Это то же самое, как переключение к приостановленному потоку, или пробуждение ОС после гибернации. С тем исключением, что вы можете проделывать это много раз подряд. После пробуждения ОС информация о гибернации уничтожается. Если этого не делать, то можно было бы восстанавливать состояние ОС с одной и той же точки. Это почти как путешествие по времени. С продолжениями вы можете себе такое позволить!

36) Ввод-вывод в языке Scheme.

Порт ввода по умолчанию связан с клавиатурой (stdin, в терминах языка Си), порт вывода — с экраном (stdout, в терминах языка Си). Эти порты по умолчанию можно переназначать.

Предикат типа «порт»:

```
(port? x) → #t или #f
```

Порты по умолчанию:

```
(current-input-port) → port
(current-output-port) → port
```

Создание порта:

```
(open-input-file "имя файла") → port
(open-output-file "имя файла") → port
```

Предусловие для `open-output-port`: файл существовать не должен (иначе ошибка).
После использования порты, связанные с файлами, нужно закрывать:

```
(close-input-port port)
(close-output-port port)
```

Временное перенаправление портов:

```
(with-output-to-file "имя файла" proc)  ≡ (proc)
(with-input-from-file "имя файла" proc) ≡ (proc)
```

Здесь `proc` — процедура без параметров, во время выполнения этой процедуры порты вывода и ввода, соответственно, по умолчанию будут перенаправлены. Возвращаемое значение у этих процедур то же, что у вызванной процедуры.

```
(with-output-to-file "D:\\test.txt"
        (lambda ()
                (display 'hello)))
```

В файл `D:\test.txt` будет записана строчка `hello`.

Открытие портов с автоматическим закрытием:

```
(call-with-input-file "имя файла" proc)  ≡ (proc port)
(call-with-output-file "имя файла" proc) ≡ (proc port)
```

Здесь процедура proc будет принимать порт в качестве параметра:

```
(call-with-output-file "/home/user/test2.txt"
        (lambda (port)
                (display 'hello port)))
```

Порт, переданный в процедуру, будет закрыт при завершении вызова самой процедуры.

37) Средства для метапрограммирования языка Scheme.

Процедура `eval` (eval expression enviroment) Пример:

```
(define foo (list '+ 1 2))
(eval foo (interaction-environment)) # -> 3
```

Выполняет кусок кода написанный в expression. Позволяет выполнять программы "на лету".

38) Хвостовая рекурсия и ее оптимизация интерпретатором языка Scheme.

Хвостовой вызов — вызов, который является последним, результат этого вызова становится результатом работы функции.

В языке Scheme заложена оптимизация хвостового вызова, т.н. оптимизация хвостовой рекурсии. Фрейм стека (см. лекцию про продолжения) вызывающей процедуры замещается фреймом стека вызываемой процедуры.

Если хвостовой вызов является рекурсивным, фреймы стека не накапливаются. Хвостовая рекурсия в языке Scheme эквивалента итерациипо вычислительным затратам

39) Основные управляющие конструкции языка Scheme.

`define` , `define-syntax` , `lambda` , `if` , `set!` , `cond` , `begin` , `do`

40) Ассоциативные списки.

Ассоциативный список — это способ реализации ассоциативного массива (т.е. структуры данных, отображающей ключи на значения) при помощи списка, это список пар (cons-ячеек), где в car находится ключ, а в cdr — связанное значение. Частный случай — список списков, где car'ы — ключи, а хвосты — значения. Чаще всего это частный случай и встречается, т.к. правильные списки просто удобнее.

Пример:

```
'((a 1) (b 2) (c 3))
```

отображает имена на некоторые числа.

Процедура `assoc` принимает ключ и ассоциативный список и возвращает первый элемент с заданным ключом:

```
(assoc 'b '((a 1) (b 2) (c 3))) → (b 2)
(assoc 'x '((a 1) (b 2) (c 3))) → #f
```

Поиск Ассоц.	список	Предикат
member	assoc	equal?
memv	assv	eqv?
memq	assq	eq?

equal? – глубокое сравнение
eqv? – логическое сравнение значений
eq? – сравнение по указателю, только для символов и уникальных объектов

41) Точечные пары и списки в языках семейства Lisp.

Точечная пара - структура, имеющая 2 поля: голову и хвост. Конструируется с помощью процедуры `cons`. Получение головы пары - `car`, хвоста - `cdr`.

Пары используются для создания списков. **Список** - пары, имеющая вложенные пары в хвосте, последняя пара имеет в хвосте нулевой элемент. К примеру: список (a b c d e) представляется как (a. (b. (c. (d . (e . ())))).

Для создания списков можно использовать процедуру `list`.

И список и пара, удовлетворяют предикату `pair?`

Так же существуют **неправильные списки** - список, у которого вместо последнего нулевого элемента идет ненулевой элемент. Получить такой список можно при помощи процедуры `append`, которая служит для склеивания пар и списков.

Пример:

```
(append '(a b) '(c . d))  ;    ===>  (a b c . d)
(append '(x) '(y))        ;    ===>  (x y)
```

42) Командный интерпретатор Bash: общая характеристика языка, пример сценария командной оболочки.

Bash (Bourne Again Shell) — это командный интерпретатор (shell) и язык сценариев, используемый в Unix-подобных операционных системах. Он является усовершенствованной версией **sh (Bourne Shell)** и совместим с ним, но также включает дополнительные возможности.

динамическая типизация (типы данных - строка и массив), слабая типизация - $(()), скриптовый интерпретируемый язык, компиляторов нет, предназначение - управление запуском команд unix-оболочки

Bash интерпретируемый язык

```
#!/bin/bash
read -p "Введите число: " num
if (( num % 2 == 0 )); then
    echo "Число $num — чётное."
else
    echo "Число $num — нечётное."
fi
```

43) Действия программиста для создания сценария («скрипта») на интерпретируемом языке программирования, предназначенного для запуска из командной оболочки UNIX-подобной операционной системы.

В начале файла следует указать шебанг-паттерн с путем к интерпретатору языка, на котором написан скрипт. Например:

```
#!/usr/bin/env python
```

Далее нужно сделать текстовый файл со скриптом исполняемым. В этом поможет команда `chmod`

```
chmod +x script.sh
```

44) Стандартные потоки ввода-вывода, аргументы командной строки. Перенаправление ввода-вывода в командной оболочке Bash, использование конвейеров (pipes).

В bash есть встроенные файловые дескрипторы: 0 (stdin), 1 (stdout), 2 (stderr).

- `stdin` - стандартный поток ввода
- `stdout` - стадартный поток вывода
- `stderr` - стандартный поток ошибок

Для операций с этими дескрипторами, существуют специальные символы: > (перенаправление вывода), < (перенаправление ввода).

Примеры:

- `0<filename` или `<filename` - перенаправление ввода из файла filename
- `1>filename` или `›filename` - перенаправление вывода в файл filename, файл перезаписывается поступающими данными
- `1>>filename` или `>>filename` - перенаправление вывода в файл filename, данные добавляются в конец файла
- `2>filename` - перенаправление стандартного вывода ошибок в файл filename
- `2>>filename` - перенаправлени стандартного вывода ошибок в файл filename, данные добавляются в конец

Конвейер передает вывод предыдущей команды на ввод следующей или на вход командного интерпретатора. Метод часто используется для связывания последовательности команд в единую цепочку. Конвейер обозначается следующим символом: |.

Пример (grep в качестве фильтра для стандартного потока ввода):

```
cat filename | grep something
```

45) Файловая система, путь к файлу и атрибуты файла. Основные команды оболочки для работы с файлами и папками.

Для каждого процесса существует своего рода глобальная переменная — т**екущая папка**. Как правило, **текущая папка** — это папка, которая была текущей в родительском процессе на момент запуска дочернего. Но процесс при желании может эту папку сменить.

Путь к файлу может быть **аблолютным** или **относительным**. Абсолютный путь к файлу указывается относительно корня операционной системы, относительный — относительно текущего каталога.

В путях можно использовать такие синонимы, как «.» и «..». Знак «.» является синонимом текущей папки, знак «..» — родительской папки. Можно считать, что в каждой папке находится папка «.», которая является синонимом для неё же самой и папка «..», которая является синонимом для родительской папки (ссылка на родительскую папку). В корневой папке «..» ссылается на неё же саму.

1. **Файловая система, путь к файлу и атрибуты файла. Основные команды оболочки для работы с файлами и папками**

Файловая система — способ хранения информации в долговременной памяти компьютера (жёсткие диски, флешки, ...) и соотвеющее API операционной системы.

Путь к файлу — способ указания конкретного файла в файловой системе.

Исполнимые файлы в UNIX-подобных ОС отличаются от обычных флагом исполнимости. У каждого файла есть три набора флагов (*атрибутов*) rwxrwxrwx, r — доступ на чтение, w — доступ на запись, x — доступ на исполнение. Первая группа — права владельца файла, вторая — права группы пользователей, владеющих файлом, третья — права для всех остальных.

Права доступа типичного файла: rw-r–r–, т.е. владелец может в файл писать, все остальные — только читать.

Права доступа: –x–x–x — файл нельзя прочитать, но можно запустить.

Установка и сброс атрибутов выполняется командой chmod:

```
chmod +x prog        # добавить флаг исполнимости
chmod +w file.dat    # разрешить запись
chmod -w file.dat    # запретить запись
chmod go-r file.dat  # запретить чтение (r) группе (g) и всем остальным (o)
```

Источник < https://bmstu-iu9.github.io/scheme-labs/lect14.html>

Основные команды:

touch file — создать файл.
realpath file — узнать абсолютный путь к файлу.
stat file1 — получение информации о «file1» (размер файла, дата создания файла и т. д.) и проверка существования файла.
cat > file — запись в файл.
cat file — чтение файла.
echo текст >> file — дописать в файл текст.
find file — поиск файла.
mcedit file — редактирование файла (также можно использовать редакторы Nano, Vim и другие).
cat file1 file2 > file12 — объединение файлов.
sh filename — запустить файл со сценарием Bash.
./filename — запустить исполняемый файл.
cp file1 file2 — копировать файл «file1» с переименованием на «file2». Произойдёт замена файлов, если элемент с таким же названием существует.
mv file1 file2 — переименовать файл «file1» в «file2».
mv filename dirname — переместить файл «filename» в каталог «dirname».
less filename — открыть файл в окне терминала.
file filename — определение типа файла.
head filename — вывод нескольких начальных строк из файла на экран (построчное чтение файла). По умолчанию строк 10.

Источник < https://eternalhost.net/base/vps-vds/bash-rabota-s-faylami>

46) Общая характеристика языка Python, типизация и система типов.

Python – высокоуровневый интерпретируемый язык программирования, ориентированный на повышение производительности разработчика и читаемости кода, построенный на идеях императивного, объектно-ориентированного и функционального программирования. Язык создан Гвидо ван Россумом в 1989 году. Синтаксис ядра Python минималистичен. В то же время стандартная библиотека включает большой объём полезных функций. Основные архитектурные черты - динамическая строгая неявная типизация, автоматическое управление памятью, механизм обработки исключений, поддержка многопоточных вычислений и удобные высокоуровневые структуры данных. Код в Python организовывается в функции и классы, которые могут объединяться в модули (они в свою очередь могут быть объединены в пакеты).

Python - интерпретируемый язык с динамической системой типов, со строгой неявной типизацией.

47) Понятие объекта. Создание и использование объектов в языке Python.

В Python **всё является объектом**: числа, строки, списки, функции, классы и даже модули.
Каждый объект имеет:

- **Тип** (класс, определяющий его поведение)
- **Идентификатор** (уникальный адрес в памяти)
- **Значение** (данные, которые он хранит)

```
x = 42
print(id(x))   # 4385365440
print(type(x)) # <class 'int'>
print(x)       # 42
```

Создание собственных объектов (классы и экземпляры)

```
class Car:
    def __init__(self, brand, year):
        self.brand = brand
        self.year = year

    def info(self):
        return f"{self.brand}, {self.year}"
```

```python
car1 = Car("Toyota", 2020)
car2 = Car("BMW", 2022)

print(car1.info())  # Toyota, 2020
print(car2.info())  # BMW, 2022
```

48) Применение функций высшего порядка для обработки последовательностей на языке Python.

- `map()` — Применение функции ко всем элементам
- `filter()` — Фильтрация элементов
- `reduce()` — Свёртка последовательности
- `sorted()` с key — Сортировка по сложному критерию
- `all()` и `any()` — Проверка условий

49) Лексические замыкания (на примере) в языке Python и связанные переменные.

Замыкание (closure) — это функция, **которая запоминает переменные из своей внешней области видимости**, даже если вызывается вне неё.

```python
def outer():
    count = 0
    def inner():
        nonlocal count
        count += 1
        print(count)
    return inner

counter = outer()
counter()  # 1
counter()  # 2
counter()  # 3
```

50) Особенности логических операций в языке программирования Python.

Логические операции `and` и `or` являются примером нестрогих вычислений. Также они возвращают не `True` / `False`, а последнее проверенное значение.

`not` всегда возвращает `True` или `False`

51) Обработка исключений в языке Python.

```
try:
    risky_operation()
except ExceptionType:
        pass
```

Примеры исключений:

- ZeroDivisionError — деление на 0
- TypeError — операция с несовместимыми типами
- IndexError — выход за границы списка
- KeyError — отсутствующий ключ в словаре

52) Определение класса и создание объекта класса на языке Python.

Класс определяется с помощью ключевого слова class .
Пример определения класса и создание объекта класса:

```
class Point:
    def __init__ (self, x = 0.0, y = 0.0, z = 0.0):
        self.x = x
        self.y = y
        self.z = z

p = Point(1, 2, 3)
```

53) Средства метапрограммирования языка Python.

В отличие от Scheme вместо символьного типа используются строки.

```
s = '2 + 2'
x = eval(s)
print (x) → 4
```

С помощью `eval` можно также определить функцию и вызвать ее по имени.

```
f = eval( 'lambda a, b: a ** 2 + b ** 2')
print (f(2,3)) => 13
```

А еще можно использовать `exec`.

```
exec("def g(a, b):\n return a + b")
print(g(2,3)) => 5
```

Чем отличаются `eval` и `exec` ?

- `eval()` возвращает значение
- `exec()` выполняет код и игнорирует возвращаемое значение (возвращает `None` в Python 3, а в Python 2 вовсе является высказыванием, поэтому ничего не возвращает)

Можно использовать `compile()` , если какой-то код требуется выполнить несколько раз. При этом в качестве одного из аргументов следует передать режим выполнения - `exec` или `eval`.

```
code = compile('print("have a great day")', '', 'exec')
exec(code)
```

К слову, как `eval`, таки `exec` сами вызывают тот же `compile()`.

54) Особенности присваивания, копирования и передачи в функцию объектов в языке Python.

Присваивание (=) копирует только ссылку
Когда вы присваиваете переменной объект, **копируется только ссылка, а не сам объект**.

Для копирования объектов используется модуль `copy`. При этом различают поверхностную копию и глубокую.

- Поверхностная копия (`copy.copy()`) создает новый составной объект и затем вставляет в него ссылки на объекты, находящиеся в оригинале.
- Глубокая копия (`copy .deepcopy()`) создает новый составной объект, а затем рекурсивно вставляет в него копии объектов, находящиеся в оригинале.

В Python **передача аргументов всегда по ссылке**.

```
def modify(lst):
    lst.append(99)

numbers = [1, 2, 3]
```

```
modify(numbers)
print(numbers)  # [1, 2, 3, 99]
```