Подготовка к экзамену по основам информатики (1 семестр)

David edited this page 12 days ago · 26 revisions

Экзамен включает в себя вопросы из разных категорий и несколько несложных практических задач на языках Scheme и Python.

Содержание

1 вопросы: определения понятий, теория

1) Понятие данных.

Данные - поддающееся многократной интерпретации представление информации в формализованном виде, пригодном для передачи, связи или обработки.

2) Понятие программы, алгоритма.

Программа - последовательность инструкций, описывающая выполнение некоторых задач на компьютере.

Алгоритм - набор инструкций, описывающих порядок действий некоторого исполнителя для решения задачи за конечное число действий.

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

Тип данных - это

- некоторое множество значений
- набор операций, которые можно принменить к этим значениям
- способ хранения значений в памяти ЭВМ

Тип данных - это относительно устойчивая и независимая совокупность элементов рассматриваемого множества.

Система типов языка программирования - набор правил, определяющих свойства конструкций языка: переменных, значений, выражений и т.д..

Переменная - поименованная либо адресуемая каким-либо способом область памяти, адреса которой может быть использованы для осуществления доступа к данным и изменения их значения в ходе выполнения программы

Типизация - контроль типов. Определение типа в программе может быть явным (Си, Паскаль) или неявным (Scheme).

Контроль типов может осуществляться *во время компиляции* (тогда говорят о статической типизации) или *во время выполнения программы* (в случае динамической типизации).

Также типизацию разделяют по строгости на *строгую* и *слабую*.

Типы бывают *встроенные* и *определяемые программистом*. И те, и другие могут быть простыми и составными.

Полиморфный тип - тип, представляющий собой набор типов. Пример: `number` в Scheme (`integer` , `real` , `rational` , `complex`).

4) Важнейшие парадигмы программирования и их отличительные черты.

Парадигма программирования - способ представления процесса вычислений, то есть способ понимания программиста процесса произведения вычисления.

Вычисление - математическое преобразование исходного потока информации в выходной со структурой, отличной от исходной.

Основные парадигмы программирования.

Императивное программирование (как?) описывает процесс вычислений в виде последовательности инструкций, изменяющих состояние программы (Паскаль, Си).

+:

- близость к архитектуре компьютера

-:

- многосложность программ
- длинные программы
- интенсивное использование деструктивного присваивания, порождающее дополнительные ошибки

К императивному программированию относят процедурное и структурное.

Процедурное программирование - оформление логически целостных часто используемых частей программы в виде именованных подпрограмм, к которым можно обращаться для выполнения повторяющихся операций с повторяющимися данными.

Структурное програмирование - представление программы в виде иерархической структуры блоков. Любая программа строится из трех базовых управляющих структур: последовательность, ветвление и цикл.

Декларативное программирование (что?) предполагает описание логики вычислений или отношений между объектами без явного указания последовательности действий.

Функциональное программирование - парадигма программирования и раздел дискретной математики, в котором вычисления рассматриваются как вычисление значений функций.

+:

- простота кода
- отладка частей программы независимо друг от друга

-:

- меньшая эффективность

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 - тип данных, который рассматривается независимо от контекста и реализации в программе.

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.

9) Мемоизация результатов вычислений.

Мемоизация - сохранение результатов выполнения функций для предотвращения повторных вычислений. Это один из способов оптимизации, применяемый для увеличения скорости выполнения компьютерных программ.

Мемоизация процедуры для вычисления факториала в Scheme:

```
(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.

Нестрогие вычисления означают, что аргументы не вычисляются до тех пор, пока их значение не используется в теле функции. *Примеры:* `or` и `and` в языке Scheme.

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.

11) Способы реализации языка программирования высокого уровня.

Языки программирования могут быть реализованы как *компилируемые* или *интерпретируемые*.

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

Компилятор - программа, выполняющая компиляцию.

Компиляция - трансляция программы, составленной на исходном языке высокого уровня, в эквивалентную программу на низкоуровневом языке, близком к машинному коду.

Процесс компиляции происходит в несколько этапов:

1. **Лексический анализ** (им занимается лексер/сканер) - преобразование исходной последовательности символов в последовательность лексем (токенов). *Лексема* - последовательность допустимых символов языка программирования, имеющая смысл для транслятора. *Лексемами* называют минимальные значимые единицы текста программы.
2. **Синтаксический анализ** (им занимается синтаксический анализатор / парсер) - процесс сопоставления линейной последовательности лексем и его формальной грамматики.

Синтаксический анализ - разбор исходной строки символов в соответствии с правилами формальной грамматики в структуру данных (дерево разбора).
3. **Семантический анализ** - анализ вычислений. Обработка дерева разбора для установления смысла: например, привязка идентификаторов к их декларациям, типам, определение типов выражений и т.д.
4. **Оптимизация** (необязательный этап) - удаление излишних конструкций и упрощение кода с сохранением его смысла.
5. **Генерация кода** - машинного кода или байт-кода.

Байт-код - промежуточное представление, в которое может быть переведена компьютерная программа. Это компактное представление программы, уже прошедшей синтаксический и семантический анализ.

Интерпретатор - программа, выполняющая интерпретацию.

Интерпретация - пооператорный (покомандный) анализ, обработка и тут же выполнение программы или запроса.

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

Назначение: преобразование исходной последовательности символов в последовательность токенов.

Входные данные: последовательность символов программы, записанной на исходном языке.

Выходные данные: последовательность токенов (лексем).

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: процесс сопоставления линейной последовательности лексем и его формальной грамматики.

Входные данные: последовательность токенов (лексем).

Выходные данные: дерево разбора.

15) Формальная грамматика, терминальные символы, нетерминальные символы.

Формальная грамматика - описание языка, то есть подмножество слов, которое можно составить из некоторого конечного алфавита.

<T, N, P, S>:

- T - конечное множество терминальных символов (терминалов)
- N - множество нетерминальных символов.
- P - множество правил или продукций (правила имеют вид: x -> <a0, a1, ...>, где x - нетерминальный символ, a - терминальные символы, причем выполняется a принадлежит объединению T с N)
- S - начальное правило (аксиома)

Терминальный символ (*терминал*) - это символ, присутствующий в словах языка и имеющий конкретное неизменяемое значение. (Терминал уже не может быть разобран на

составляющие.) Терминальный символ - символ, принадлежащий множеству терминальных символов языка.

Нетерминальный символ (*нетерминал*) - это объект, обозначающий какую-либо сущность языка, но не имеющий конкретного символьного представления.

16) БНФ.

БНФ - форма Бэкуса-Наура. БНФ - формальная система описания синтаксиса, в которой одни синтаксические категории последовательно определяются через другие категории. Используется для описания контекстно-свободных формальных грамматик.

x -> a1, a2, a3 ...

x - цепочка символов. Цепочка в грамматике может быть пустой, тогда она обозначается ε.

- ::= - обозначение стрелки
- . - конец правила
- a (строчная) - терминальный символ
- <A> - нетерминальный символ

Пример:

> <S> :: = <A> a | b
>
> <A> ::= <S>c | .

Еще примеры:

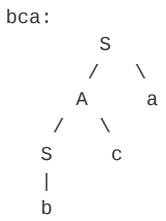
> Digit ::= "0" | "1" | ... |"9".
>
> Letter ::= "A" | "B" | ... | "Z".
>
> Number ::= Number Digit | Digit.
>
> Var ::= Letter | Var Letter | Var Digit .

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.

> <S> ::= <A> a | b.
>
> <A> = <S> c .

Одно из возможных деревьев разбора:



Корень дерева - аксиома. Листья помечены терминальными символами. Вершины помечены нетерминальными символами.

Множество предложений, полученных из всех синтаксических деревьев есть язык этой грамматики.

Задача синтаксического анализа - выяснить, является ли некоторая последовательность символов предложением языка, заданного формальной грамматикой.

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 - алгоритм синтаксического анализа, реализуемый путём последовательного вызова взаимно рекурсивных процедур, каждая из которых соответствует одному правилу контекстно-свободной грамматики LL(k)

Сложность: O(n)

Пример грамматики для синт. анализа:

```
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 аттрибуты (свойства) и методы (возвращают или изменяют состояние объекта).

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

Модульное тестирование - процесс в программировании, позволяющий проверить на корректность отдельные модули исходного кода программы. Идея в том, чтобы *писать тесты* для каждой нетривиальной функции или метода.

Разработка через тестирование (TDD) - сначала пишется тест, покрывающий желаемое изменение, затем пишется код, который проходит данный тест.

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 операций.

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.

Разбор осуществляется методом рекурсивного спуска. Если исходная последовательность не соответствует грамматике, парсер возвращает `#f` .

При построении дерева разбора соблюдается общепринятая ассоциативность бинарых операторов: левую для сложения, вычитания, умножения и деления и правую для возведения в степень. Вложенность списков должна однозначно определять порядок вычисления значения выражения.

- *Преобразователь дерева разбора в выражение на Scheme* Дерево, возвращенное в результате синтаксического анализа, преобразуется в выражение на языке Scheme.

Полученное выражение пригодно для вычисления его значения интерпретатором языка Scheme.

2 вопросы: язык программирования Scheme + разное

1) Основные постулаты языков программирования семейства Lisp.

- единство кода и данных
- все - это список (упорядоченный набор элементов)
- выражения - тоже список
- любое выражение возвращает значение

2) Общая характеристика языка Scheme.

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

3) Типизация и система типов языка Scheme.

Система типов - строгая, динамическая.

4) 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

5) 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

```
(eqv? '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 xyz, 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 == case-insensitive
(char-ci=? #\a #\A) =>  #t

(char-downcase #\A) =>  #\a
(char-upcase #\a)   =>  #\A
```

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

Составные типы данных строятся путем соединения значений других типов данных.

- **Строки**

Строки это последовательности, состоящие из characters:

```
(string #\h #\e #\l #\l #\o)
=>  "hello"

(define greeting "Hello; Hello!")
(string-ref greeting 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**

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 проихвольных 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
```

The elements of a dotted pair can be replaced by the mutator procedures set-car! and set-cdr!:
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 #\l #\l #\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
```

7) Символьный тип в языке Scheme и его применение.

character (char) -- символьный тип языка ским, предназначеный для представления 1 печатного символа(символьный литерал) ((?)символа unicode(на самом деле нет)).

Записываются в следующей нотации: `#\<character>` или `#\<character name>` .

Примеры:

```
#\a ; строчная буква
#\A ; заглавная буква
#\( ; открывающая скобка
#\  ; символ пробел
#\space ; более предпочтительный вариант символа пробела
#\newline ; символ перехода на новую строку
```

Предикат типа:

```
(char? obj)
```

Основные процедуры для работы с char:

```
(char=? char1 char2) ; а также char<?, char>? и т.п.
(char->integer char) ; возвращает целочисленное представление символа
(integer->char n) ; возвращает символ, соответствующий целому числу n
(char-upcase char) ; возвращает соответствующий символ заглавной буквы (если возможн
(char-downcase char) ; возвращает соответствующий символ строчной буквы,
                     ; работает аналогично char-upcase
```

Тип symbol

Основным отличием типа *symbol* является то, что два "символа" (здесь и далее "символ" == *symbol*) считаются одинаковыми тогда и только тогда, когда их написание совпадает (с точностью до регистра). По этой причине в большинстве реализаций Scheme "символы" используются для представления идентификаторов различных объектов. Однако, это далеко не единственное их назначение. Так, например, "символы" можно использовать аналогично перечисляемым типам в Pascal.

```
(eq? 'mISSISSIppi 'mississippi) ; ===> #t
```

Правила записи "символов" полностью совпадают с правилами записи идентификаторов.

Примеры: q, soup, v17a, +

Предикат типа: `(symbol? obj)`

Примеры использования:

```
(symbol? 'foo)         ;         ===>  #t
(symbol? (car '(a b))) ;         ===>  #t
(symbol? "bar")        ;         ===>  #f
(symbol? 'nil)         ;         ===>  #t
(symbol? '())          ;         ===>  #f
(symbol? #f)           ;         ===>  #f
```

Основные процедуры:

```
(symbol->string symbol) ; возвращает "имя" "символа" в виде строки (регистр зависит
; Пример:
(symbol->string 'Martin) ===>  "martin"
(string->symbol string) ; возвращает "символ", "именем" которого является аргумент
; Пример:
(eq? 'bitBlt (string->symbol "bitBlt"))  ===>  #f
          (eq? 'JollyWog
            (string->symbol
             (symbol->string 'JollyWog)))  ===>  #t
```

8) 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. Математики такую функцию чаще называют оператором, например, оператор взятия производной или оператор интегрирования.

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 (make-val-list v0 dv vN)
  (define (make-list v)
    (if (= v vN)
    (cons v '())
    (cons v (make-list (+ v dv)))))
  (make-list v0))

(define A-list (make-val-list 1 1 3))
(define x-list (make-val-list 0.2 0.1 1.2))

(define (y A)
  (lambda (x) (- (* A x) (tan (* pi (/ x 4))))))

(define y-list (map (lambda (A)
             (map (y A) x-list))(начало - см. вопрос 9)*
           A-list))

(display (map (lambda (ls) (apply max ls)) y-list))
```

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

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

```
(define test-list '(-2 1 7 0 -5 4))
(map abs test-list)                 ; вернет список абсолютных величин значений спи
```

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

```
(max -2 1 7 0 -5 4)  ; \
(+ -2 1 7 0 -5 4)    ; _вернут максимальное значение и сумму значений переданных в к
```

однако попытка вызвать их таким образом:

```
(max test-list)      : \
(+ test-list)        ; _выдаст ошибку.
```

Это не очень удобно, когда, например, в ходе работы программы генерируется некий список значений, из которого потом нужно выбрать максимальное значение, или сумму подсчитать. Тут-то и приходит на помощь функция apply, которая как бы раскрывает список, т. е. выражения:

```
(apply max test-list)  ; \
(apply + test-list)    ; _эквивалентны выражениям
(max -2 1 7 0 -5 4)    ; \
(+ -2 1 7 0 -5 4)      ; _соответственно.
```

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.

9) 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).

Замыкание — это особый вид функции. Она определена в теле другой функции и создаётся каждый раз во время её выполнения. Синтаксически это выглядит как 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.

Пример:

```
(define (make-adder n)       ; возвращает замкнутое лямбда-выражение
  (lambda (x)                ; в котором x - связанная переменная,
    (+ x n)))                ; а n - свободная (захваченная из внешнего контекста)

(define add1 (make-adder 1)) ; делаем процедуру для прибавления 1
(add1 10)                    ; возвращает 11

(define sub1 (make-adder -1)); делаем процедуру для вычитания 1
(sub1 10)                    ; возвращает 9
```

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). Лексические переменные замыкания отличаются от глобальных переменных тем, что они не занимают глобальное пространство имён. От переменных в объектах они отличаются тем, что привязаны к функциям, а не объектам.

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

Другими словами, если есть переменная, объявленная где-то в программе, и есть функция, которая имеет доступ к этой переменной, то такая переменная будет называться свободной.

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

Макросы let и let* через анонимные процедуры:

```
(use-syntax (ice-9 syncase))

(define-syntax my-let
  (syntax-rules ()
    ((_ ((a b) ...) body ...)
     ((lambda (a ...) body ...) b ...))))

(define-syntax my-let*
  (syntax-rules ()
    ((_ ((a b)) body ...)
     (my-let ((a b)) body ...))
    ((_ ((a b) (c d) ...) body ...)
     (my-let ((a b))
             (my-let* ((c d) ...)
                      body ...))))))
```

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

(начало - см. вопрос 9)

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))
```

11) 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.

Примеры:

```
(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.

Примеры:

```
(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)
```

12) Гигиенические макросы в языке Scheme.

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.

Примеры макросов:

```
(define-syntax when
  (syntax-rules ()
    ((_ condition expr ...)
      (if condition
          (begin expr ...)))))

(define-syntax unless
  (syntax-rules ()
    ((_ condition expr ...)
     (when (not condition) expr ...))))
```

13) 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!

Рассмотрим в качестве примера выражение:

`(+ 3 (* 4 (call/cc r)))` Исходное выражение можно раскрыть до:

`(+ 3 (* 4 (r (escaper (lambda (▫) (+ 3 (* 4 ▫)))))))`

Допустим, r -- это `(lambda (continuation) 6)`; тогда выражение преобразуется в:

`(+ 3 (* 4 ((lambda (continuation) 6) (escaper (lambda (▫) (+ 3 (* 4 ▫)))))))`

Продолжение (и функция выхода) не используются, поэтому результат вызова: `((lambda (continuation) 6) (escaper (lambda (▫) (+ 3 (* 4 ▫)))))`

равен 6, а результат всего выражения в целом равен 27 -- (3 + 4 * 6).

Но если бы за r мы приняли `(lambda (continuation) (continuation 6))`, то получилось бы следующее выражение:

```
(+ 3 (* 4 ((lambda (continuation) (continuation 6)) (escaper (lambda (▫) (+ 3 (* 4 ▫)))))))
```

В результате применения продолжения к аргументу 6 получаем:

```
((escaper (lambda (▫) (+ 3 (* 4 ▫)))) 6)
```

Как уже говорилось, escaper возвращает функцию выхода, которая сбрасывает свой контекст (как следствие, + и *, ожидающие результата вызова escaper, игнорируются). Хотя результатом вычисления снова оказалось 27, процесс получения результата был существенно иным.

Ещё примеры:

```
(define call/cc call-with-current-continuation)

(define return #f)
(+ 2 (call/cc
      (lambda (cont)
        (set! return cont)
        1))) ; ->3
(return 5) ;-> 7

(define (map1/x xs)
  (call/cc (lambda (escape)
             (define (helper xs)
               (if (null? xs)
                   '()
                   (if (zero? (car xs))
                       (escape #f)
                       (cons (/ 1 (car xs)) (helper (cdr xs))))))
             (helper xs))))

(map1/x '(5 2 4 3)) ;->(1/5 1/2 1/4 1/3)
(map1/x '(1 2 0 3)) ;->#f

(define (search wanted? lst)
  (call/cc
   (lambda (return)
     (for-each (lambda (element)
                 (if (wanted? element)
                     (return element)))
               lst) #f)))
(search (lambda (x) (> x 0)) '(-1 -2 3 -4 5)) ;-> 3
(search (lambda (x) (> x 0)) '(-1 -2 -3 -4 -5)) ;->#f
```

14) Ввод-вывод в языке Scheme.

Для ввода и вывода в Scheme используется тип порт. R5RS определяет два стандартных порта, доступные как `current-input-port` и `current-output-port`, отвечающие стандартным потокам ввода-вывода Unix. Большинство реализаций также предоставляют `current-error-port`.

- `(write obj port)` - вывод (результат закавычивается - "123")
- `(display obj port)` - тоже вывод (результат не закавычивается)
- `(newline port)` - записывает символ конца строки в указанный порт
- `(write-char char port)` - записывает символ (не его внешнее представление) в указанный порт

Стоит отметить, что в приведенных выше процедурах можно *не указывать порт*. В таком случае вывод будет осуществляться в `current-output-port`.

- `(read port)` - считывание до пробела, символа табуляции или перехода на новую строку
- `(read-char port)` - считывание одного символа из указанного порта

- `(peek-char port)` - считывание следующего символа из порта, при этом переход на следующий символ не осуществляется

15) 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".

16) Хвостовая рекурсия и ее оптимизация интерпретатором языка Scheme.

Хвостовая рекурсия — частный случай рекурсии, при котором любой рекурсивный вызов является последней операцией перед возвратом из функции.

Пример:

```
(define (factorial n)
  (define (fac-times n acc)
    (if (= n 0)
        acc
        (fac-times (- n 1) (* acc n))))
  (fac-times n 1))
```

Создатели функционального языка Scheme, одного из диалектов Lisp, оценили важность хвостовой рекурсии настолько, что в спецификации языка предписали каждому транслятору этого языка в обязательном порядке реализовывать оптимизацию хвостовой рекурсии и описали точный набор условий, которым должна отвечать рекурсивная функция, чтобы рекурсия в ней была оптимизирована.

17) Основные управляющие конструкции языка Scheme.

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

В лексере языка Scheme выделяются только они.

18) Ассоциативные списки.

Ассоциативный массив (словарь) — абстрактный тип данных, позволяющий хранить пары вида "(ключ, значение)" и поддерживающий операции добавления пары, а также поиска и удаления пары по ключу.

Ассоциативный список в Scheme - список, в котором находятся точечные пары. По умолчанию доступны процедуры получения головы и хвоста списка (car , cdr). Основываясь на этих процедурах мы можем реализовать функции поиска элемента по ключу, добавления нового элемента и удаления элемента с заданным ключом.

Пример:

```
( (key1 . val1) (key2 . val2) (key3 . val3))
```

Уже существующие процедуры:

`alist?` - предикат типа `(assq object alist)` / `(assv object alist)` / `(assoc object alist)` - возвращает первый найденный элемент ассоцативного списка с заданным ключем или `#f` , если таковой не найден

19) 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)
```

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

Bash - усовершенствованная и модернизированная вариация командной оболочки Bourne shell. Одна из наиболее популярных современных разновидностей командной оболочки UNIX. Это командный интерпретатор, работающий, как правило, в интерактивном режиме в текстовом окне. Bash также может читать команды из файла, который называется скриптом (или сценарием).

Пример сценария:

```
#!/usr/bin/env bash
for x in one two three four five
do
echo $x
done
```

```
#!/bin/bash
#Получение тестов к задачам T-BMSTU
#Параметры: ссылка на тесты (не включая номер) и кол-во тестов
#Пример ./getTests.sh http://195.19.40.181:3386/tasks/iu9/algorithms_and_data_struct

url=$1

for ((i=1; i<=$2; i++))
do
    wget "$url/$i" # Скачивает файл по ссылке
    mv $i test$i # Переименовывает файл
    wget "$url/$i.a" # Скачивает ответы к тесту
    mv $i.a test$i-ans # Переименовывает файл
done

echo "Тесты загружены."
```

21) 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
```

22) 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
```

3 вопросы: язык программирования Python + разное

1) Общая характеристика языка Python.

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).

2) Типизация и система типов языка Python.

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

3) Способы определения функции в языке Python. Формальные и фактические аргументы, применение к аргументам, возвращаемое значение.

Функции в Python определяются 2-мя способами:

- через определение `def`
- через анонимное описание `lambda` .

Особенностью Python является то, что функция является таким же именованным объектом, как и любой другой объект некоторого типа данных, скажем, как целочисленная переменная. Пример:

```
def pow3(n):                    # 1-е определение функции
    return n * n * n
print (pow3, n)

pow3 = lambda n: n * n * n         # 2-е определение функции с тем же именем
print (pow3, n)

print (( lambda n: n * n * n ), n) # 3-е, использование анонимного описание функции
```

В определении функции фигурируют формальные аргументы. Некоторые из них могут иметь значения по умолчанию. Все аргументы со значениями по умолчанию следуют после аргументов без значений по умолчанию. При вызове функции задаются фактические аргументы. Например:

```
pow3 (5)
```

В начале идут позиционные аргументы. Они сопоставляются с именами формальных аргументов по порядку. Затем следуют именованные аргументы. Они сопоставляются по именам и могут быть заданы в вызове функции в любом порядке. Разумеется, все аргументы, для которых в описании функции не указаны значения по умолчанию, должны присутствовать в вызове функции. Повторы в именах аргументов недопустимы.

В языке Python никогда не указывается не только тип возвращаемого значения, но даже его наличие. На самом деле каждая функция возвращает значение; если функция выполняет инструкцию return, она возвращает указанное в ней значение, иначе функция возвращает специальное значение -- None.

4) Простые типы языка Python и основные операции над ними.

- None
- Логический - может принимать одно из двух значений -- True (истина) или False (ложь).
- Числа, могут быть целыми (1 и 2), с плавающей точкой (1.1 и 1.2) и даже комплексными. Над числами можно выполнять различные операции.

```
>>>11 / 2
5.5
>>> 11 // 2
5
>>> −11 // 2
−6
>>> 11.0 // 2
5.0
>>> 11 ** 2
121
>>> 11 % 2
1
```

5) Типы последовательностей в языке Python и основные операции над ними.

- Строки - последовательности символов Юникода, в Python они относятся к категории неизменяемых последовательностей, то есть все функции и методы могут лишь создавать новую строку. Над строками так же можно выполнять разные операции:

```
#Конкатенация строк
>>> S1 = 'spam'
>>> S2 = 'eggs'
>>> print(S1 + S2)
>>> 'spameggs'
#Дублирование строк
>>> print('spam' * 3)
```

```
>>> spamspamspam
#Длина строки (функция len)
>>> len('spam')
4
#Доступ по индексу
>>> S = 'spam'
>>> S[0]
's'
>>> S[2]
'a'
>>> S[-2]
'a'
```

Как видно из примера, в Python возможен и доступ по отрицательному индексу, при этом отсчет идет от конца строки.

Извлечение среза

Оператор извлечения среза: [X:Y]. X – начало среза, а Y – окончание; символ с номером Y в срез не входит. По умолчанию первый индекс равен 0, а второй - длине строки.

```
>>> s = 'spameggs'
>>> s[3:5]
'me'
>>> s[2:-2]
'ameg'
>>> s[:6]
'spameg'
>>> s[1:]
'pameggs'
>>> s[:]
'spameggs'
```

Кроме того, можно задать шаг, с которым нужно извлекать срез.

```
>>> s[::-1]
'sggemaps'
>>> s[3:5:-1]
''
>>> s[2::2]
'aeg'
```

- Списки в Python - упорядоченные изменяемые коллекции объектов произвольных типов (почти как массив, но типы могут отличаться).

  Чтобы использовать списки, их нужно создать. Создать список можно несколькими способами. Например, можно обработать любой итерируемый объект (например, строку) встроенной функцией list:

  ```
  >>> list('список')
  ['с', 'п', 'и', 'с', 'о', 'к']
  ```

  Список можно создать и при помощи литерала:

  ```
  >>> s = []  # Пустой список
  >>> l = ['s', 'p', ['isok'], 2]
  >>> s
  []
  >>> l
  ['s', 'p', ['isok'], 2]
  ```

  Как видно из примера, список может содержать любое количество любых объектов (в том числе и вложенные списки), или не содержать ничего.

  И еще один способ создать список - это генераторы списков. Генератор списков - способ построить новый список, применяя выражение к каждому элементу

последовательности. Генераторы списков очень похожи на цикл for.

```
>>> c = [c * 3 for c in 'list']
>>> c
['lll', 'iii', 'sss', 'ttt']
```

Метод	Что делает
list.append(x)	Добавляет элемент в конец списка
list.extend(L)	Расширяет список list, добавляя в конец все элементы списка L
list.insert(i, x)	Вставляет на i-ый элемент значение x
list.remove(x)	Удаляет первый элемент в списке, имеющий значение x
list.pop([i])	Удаляет i-ый элемент и возвращает его. Если индекс не указан, удаляется последний элемент
list.index(x, [start [, end]])	Возвращает положение первого элемента от start до end со значением x
list.count(x)	Возвращает количество элементов со значением x
list.sort([key = функция])	Сортирует список на основе функции
list.reverse()	Разворачивает список
list.copy()	Копия списка
list.clear()	Очищает список

- Кортежи - упорядоченные неизменяемые последовательности значений. По сути - неизменяемый список. Операции с кортежами - это, по сути, все операции над списками, только не изменяющие его (сложение, умножение на число, методы index() и count() и некоторые другие операции). Можно также по-разному менять элементы местами и так далее.
- Множества -"контейнер", содержащий не повторяющиеся элементы в случайном порядке.

Операция	Что делает
len(s)	число элементов в множестве (размер множества)
x in s	принадлежит ли x множеству s
set.isdisjoint(other)	истина, если set и other не имеют общих элементов
set == other	все элементы set принадлежат other, все элементы other принадлежат set
set.issubset(other) или set <= other	все элементы set принадлежат other
set.issuperset(other) или set >= other	аналогично
set.union(other, ...)	объединение нескольких множеств
set.intersection(other, ...) или set & other & ...	пересечение
set.difference(other, ...) или set - other - ...	множество из всех элементов set, не принадлежащие ни одному из other
set.symmetric_difference(other); set ^ other	множество из элементов, встречающихся в одном множестве, но не встречающиеся в обоих

set.copy()	копия множества
set.add(elem)	добавляет элемент в множество
set.remove(elem)	удаляет элемент из множества. KeyError, если такого элемента не существует
set.discard(elem)	удаляет элемент, если он находится в множестве
set.pop()	удаляет первый элемент из множества. Так как множества не упорядочены, нельзя точно сказать, какой элемент будет первым
set.clear()	очистка множества

- Словари - неупорядоченные коллекции произвольных объектов с доступом по ключу. Их иногда ещё называют ассоциативными массивами или хеш-таблицами.

Метод	Что делает
dict.clear()	очищает словарь
dict.copy()	возвращает копию словаря
classmethod dict.fromkeys(seq[, value])	создает словарь с ключами из seq и значением value (по умолчанию None)
dict.get(key[, default])	возвращает значение ключа, но если его нет, не бросает исключение, а возвращает default (по умолчанию None)
dict.items()	возвращает пары (ключ, значение)
dict.keys()	возвращает ключи в словаре
dict.pop(key[, default])	удаляет ключ и возвращает значение. Если ключа нет, возвращает default (по умолчанию бросает исключение)
dict.popitem()	удаляет и возвращает пару (ключ, значение). Если словарь пуст, бросает исключение KeyError. Помните, что словари неупорядочены
dict.setdefault(key[, default])	возвращает значение ключа, но если его нет, не бросает исключение, а создает ключ с значением default (по умолчанию None)
dict.update([other])	обновляет словарь, добавляя пары (ключ, значение) из other. Существующие ключи перезаписываются. Возвращает None (не новый словарь!)
dict.values()	возвращает значения в словаре

6) Средства для организации циклов в программах на языке Python.

С помощью цикла можно описать повторяющиеся действия. В Python имеются два вида циклов: цикл ПОКА (выполняется некоторое условие) и цикл ДЛЯ (всех значений последовательности). Следующий пример иллюстрирует цикл ПОКА на Python:

```
s = "abcdefghijklmnop"
while s != "":
  print s
  s = s[1:-1]
```

Оператор `while` говорит интерпретатору Python: "пока верно условие цикла, выполнять тело цикла". В языке Python тело цикла выделяется отступом. Каждое исполнение тела цикла будет называться итерацией. В приведенном примере убирается первый и последний

символ строки до тех пор, пока не останется пустая строка.

Для большей гибкости при организации циклов применяются операторы break (прервать) и continue (продолжить). Первый позволяет прервать цикл, а второй - продолжить цикл, перейдя к следующей итерации (если, конечно, выполняется условие цикла). Цикл ДЛЯ выполняет тело цикла для каждого элемента последовательности. В следующем примере выводится таблица умножения:

```
for i in range(1, 10):
  for j in range(1, 10):
    print "%2i" % (i*j),
  print
```

Здесь циклы for являются вложенными. Функция range() порождает список целых чисел из полуоткрытого диапазона [1, 10).

7) Применение функций высшего порядка для обработки списков на языке Python.

В Python есть функции, одним из аргументом которых являются другие функции: map(), filter(), reduce(), apply().

Функция `map()` позволяет обрабатывать одну или несколько последовательностей с помощью заданной функции:

```
>>> list1 = [7, 2, 3, 10, 12]
>>> list2 = [-1, 1, -5, 4, 6]
>>> map(lambda x, y: x*y, list1, list2)
[-7, 2, -15, 40, 72]
```

Функция `filter()` позволяет фильтровать значения последовательности. В результирующем списке только те значения, для которых значение функции для элемента истинно:

```
>>> numbers = [10, 4, 2, -1, 6]
>>> filter(lambda x: x < 5, numbers)     # В результат попадают только такие элемент
[4, 2, -1]
```

Для организации цепочечных вычислений в списке можно использовать функцию `reduce()`. Например, произведение элементов списка может быть вычислено так (Python 2):

```
>>> numbers = [2, 3, 4, 5, 6]
>>> reduce(lambda res, x: res*x, numbers, 1)
720
```

В Python 3 встроенной функции `reduce()` нет, но её можно найти в модуле `functools`.

Функция `apply()` для применения другой функции к позиционным и именованным аргументам, заданным списком и словарем соответственно (Python 2):

```
>>> def f(x, y, z, a=None, b=None):
...     print x, y, z, a, b
...
>>> apply(f, [1, 2, 3], {'a': 4, 'b': 5})
1 2 3 4 5
```

8) "Списковые выражения" ("списковые включения", list comprehensions) в языке Python (на примерах).

Списковое включение -- наиболее выразительное из функциональных средств Python. Например, для вычисления списка квадратов положительных целых чисел, меньших 10, можно использовать выражение:

```
l = [x**2 for x in range(1,10)]
```

Списковые включения могут использовать вложенные итерации по переменным:

```
[(x, y) for x in range(1, 10) for y in range(1, 10) if x % y == 0]
```

В языке Python есть и выражения-генераторы, которые имеют схожий со списковыми включениями синтаксис, но возвращают итератор. Сумма чётных чисел:

```
sum((n for n in range(1, 10000) if n % 2 == 0))
```

Синтаксис прост. Все выражение записывается в квадратных скобках. Сначала идет выражение, которое будет задавать элементы списка, потом -- цикл с помощью которого можно изменять выражение. Обе части могут быть сколь угодно сложными.

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

Замыкание -- функция, в теле которой присутствуют ссылки на переменные, объявленные вне тела этой функции в окружающем коде и не являющиеся её параметрами. Говоря другим языком, замыкание -- функция, которая ссылается на свободные переменные в своём контексте.

Замыкание -- это особый вид функции. Она определена в теле другой функции и создаётся каждый раз во время её выполнения. Синтаксически это выглядит как 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.

Пример простого замыкания:

```
# Реализация с помощью именованных функций:
def make_adder(x):
    def adder(n):
        return x + n # захват переменной "x" из внешнего контекста
    return adder

# То же самое, но через безымянные функции:
make_adder = lambda x: (
    lambda n: x + n
)

f = make_adder(10)
print f(5) # 15
print f(-1) # 9
```

Свободная переменная -- переменная, которая:

- встречается в теле функции или предложения, но которая не является параметром этой функции,
- и/или определена в месте, находящимся где-то за пределами функции.

Другими словами, если есть переменная, объявленная где-то в программе, и есть функция, которая имеет доступ к этой переменной, то такая переменная будет называться свободной. Иногда её называют глобальной переменной.

Связанные переменные -- это аргументы функции. То есть для функции они являются локальными.

10) Особенности логических операций в языке программирования Python.

В языке Python операторы and и or выполняют булевы операции, но они не возвращают булевы значения: результатом всегда является значение одного из операндов.

```
>>> 'a' and 'b'         #(1)
'b'
>>> '' and 'b'          #(2)
''
>>> 'a' and 'b' and 'c' #(3)
'c'
```

(1) При использовании оператора and, значения вычисляются в булевом контексте слева напрво. Значения 0, '', [], (), {} и None являются ложью, все остальное является истиной[3]. Если у and оба операнда являются истиной, результатом будет последнее значение. В данном случае вычисляется выражение 'a', которое является истиной, затем 'b', которое также является истиной, и возвращается 'b'.

(2) Если какой-либо из операндов является ложью, результатом будет первое такое значение. В данном случае '' -- первое значение, являющееся ложью.

(3) Все значения являются истиной, так что в результате мы получаем последнее -- 'c'.

```
>>> 'a' or 'b'          #(1)
'a'
>>> '' or 'b'           #(2)
'b'
>>> '' or [] or {}      #(3)
{}
>>> def sidefx():
...     print "in sidefx()"
...     return 1
>>> 'a' or sidefx()     #(4)
'a'
```

(1) Как и для and опреранды or вычисляются в булевском контексте слева направо. Если операнд является истиной, or немедленно возвращает результат. В данном случае 'a' -- первое истинное значение.

(2) or вычисляет выражение '', которое является ложью, затем 'b', которое является истиной, и возвращает 'b'.

(3) Если все значения являются ложью, or возвращает последнее. or вычисляет '' (ложь), , затем {} (ложь) и возвращает {}.

(4) Обратите внимание, что or вычисляет операнды до тех пор, пока не найдет истинное значение, остальное игнорируется. Это имеет значение, когда вычисление операнда дает сторонние эффекты. В данном случае функция sidefx не вызывается, так как для получения результата выражения с опереатором or достаточно того, что первый операнд, 'a', является истиной.

11) Обработка исключений в языке Python.

Исключения появляются тогда, когда в программе возникает некоторая исключительная ситуация. Например, к чему приведёт попытка чтения несуществующего файла? Или если файл был случайно удалён, пока программа работала? Такие ситуации обрабатываются при помощи исключений.

Обрабатывать исключения можно при помощи оператора `try..except`. При этом все обычные команды помещаются внутрь `try`-блока, а все обработчики исключений – в `except`-блок.

Пример:

```
try:
    favouriteNumber = int(input('введите свое любимое число'))
```

```
except:
    print('никакое это не число, я скрипт, меня не обманешь')
```

Можно также добавить блок `else` к соответствующему блоку `try..except`. Этот пункт будет выполнен тогда, когда исключений не возникает.

Если есть код, который должен выполниться в любом случае, независимо от возникновения исключений, его можно обернуть в блок `finally`.

12) Встроенный тип "словарь" и основные операции над ним в языке Python.

Словари - неупорядоченные коллекции произвольных объектов с доступом по ключу. Их иногда ещё называют ассоциативными массивами или хеш-таблицами.

Метод	Что делает
dict.clear()	очищает словарь
dict.copy()	возвращает копию словаря
(classmethod) dict.fromkeys(seq[, value])	создает словарь с ключами из seq и значением value (по умолчанию None)
dict.get(key[, default])	возвращает значение ключа, но если его нет, не бросает исключение, а возвращает default (по умолчанию None)
dict.items()	возвращает пары (ключ, значение)
dict.keys()	возвращает ключи в словаре
dict.pop(key[, default])	удаляет ключ и возвращает значение. Если ключа нет, возвращает default (по умолчанию бросает исключение)
dict.popitem()	удаляет и возвращает пару (ключ, значение). Если словарь пуст, бросает исключение KeyError. Помните, что словари неупорядочены
dict.setdefault(key[, default])	возвращает значение ключа, но если его нет, не бросает исключение, а создает ключ с значением default (по умолчанию None)
dict.update([other])	обновляет словарь, добавляя пары (ключ, значение) из other. Существующие ключи перезаписываются. Возвращает None (не новый словарь!)
dict.values()	возвращает значения в словаре

13) Ввод-вывод в языке Python.

Для ввода данных с клавиатуры используется функция `input()`. В качестве аргумента можно передать текст приглашения для ввода.

Для вывода обычно используется функция `print()`. С помощью необязательного аргумента `file` можно вывести строку в файл или один из стандартных потоков вывода.

Пример работы с файлами:

```
f = open('Paper Towns.txt', 'rt') # другие возможные режимы: wt, rb, wb, at, ab...
for line in f.readlines():
    if 'goat' in line:
        print(line)
f.close()

l = open('love.txt', 'wt')
l.write('We love Python')
l.close()
```

14) 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) # объект класса Point: точка с координатами (1, 2, 3)
```

15) Наследование классов в языке Python.

Классы могут наследовать свойства и методы от других классов. Класс, от которого мы наследуем, указывается в скобках при объявлении класса.

```
class Sphere(Point):
    def __init__(self, x = 0.0, y = 0.0, z = 0.0, r = 1.0):
        Point.__init__(self, x, y, z)
        self.r = r
    def area(self):
        return 4 * math.pi * self.r ** 2
```

Язык Python поддерживает множественное наследование.

16) Модульная организация программ на языке Python.

Python позволяет поместить классы, функции или данные в отдельный файл и использовать его в других программах. Такой файл называется модулем. Объекты из модуля могут быть импортированы в другие модули. При импорте модуля `my_module` интерпретатор ищет файл с названием `my_module.py` сначала в текущем каталоге, затем в каталогах, указанных в переменной окружения PYTHONPATH, а затем в зависящих от платформы путях по умолчанию. Импортировать модуль можно с помощью `import`.

17) Путь к файлу, понятия абсолютного и относительного пути, имени файла и расширения файла. Средства для работы с путями к файлам в стандартной библиотеке языка Python.

Путь может быть *абсолютным* или *относительным*.

Абсолютный путь — это путь, который указывает на одно и то же место в файловой системе, вне зависимости от текущей рабочей директории или других обстоятельств. Такой путь всегда начинается с корневого каталога.

Относительный путь представляет собой путь по отношению к текущему рабочему каталогу пользователя.

Имя файла — строка символов, однозначно определяющая файл в некотором пространстве имён файловой системы (ФС), обычно называемом каталогом, директорией или папкой.

Имя файла обычно состоит из двух частей, разделенных точкой:

- название (до точки, часто также называют именем)
- расширение

С путями в Python помогает работать модуль `os.path`, вложенный в свою очередь в `os`.

```
os.path.abspath(path) # абсолютный путь к файлу
os.path.isfile(path) # является ли путь файлом
os.path.isdir(path) # является ли путь директорией
os.path.getsize(path) # размер файла в байтах
s = os.stat(path)
s.st_size # размер файла в байтах
```

```
os.path.join(path1, path2...) # соединяет пути: path1/path2
os.path.split(path) # разбивает путь на кортеж (голова, хвост), где хвост - последни
# хвост никогда не начинается со слеша (если путь заканчивается со слешем, то хвост
os.listdir(path) # список файлов и директорий в папке
os.getcwd() # текущая рабочая директория
```

18) Средства для 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` .

Пример:

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

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

19) Средства для работы с аргументами командной строки и стандартными потоками ввода-вывода в стандартной библиотеке языка Python.

Аргументы командной строки можно считать из списка `sys.argv` . При этом `sys.argv[0]` - это имя скрипта.

Для более удобной работы с аргументами командной строки также можно использовать модуль `argparse` .

Со стандартными потоками ввода и вывода (`sys.stdin` , `sys.stdout` , `sys.stderr`) можно работать как с файловыми объектами (использовать read() и write()). В том числе объект одного из потоков вывода можно передать функции `print` в качестве аргумента.

```
print('lol', file=sys.stdout)
```

20) Правила оформления и документирования исходных текстов программ на языке Python. Средства языка и интерпретатора для работы с документацией.

Программы на языке Python используют строки документации. Если первая строчка после

отступа содержит строку, считается, что это строка документации.

Также предусмотрена возможность указывать типы аргументов и возвращаемого значения для документации непосредственно в коде.

```
def fn(arg: int)->int:
    return arg * 3
```

Как правило в документацию включают краткое описание функции, аргументы, которые она принимает и возвращаемое значение.

Строки документации используются при вызове help().

21) Арифметические операции в языке Python.

- x + y - сложение
- x - y - вычитание
- x * y - умножение
- x / y - деление (результатом будет вещественное число)
- x // y - деление нацело
- x % y - остаток от деления
- - x - смена знака у числа
- abs(x) - взятие модуля
- x ** y - возведение x в степень y

22) Особенности копирования объектов в языке Python.

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.

4 вопрос: Задачи

Примеры

1) На языке Python напишите возможную реализацию встроенной функции filter.

Решение:

```
def my_filter(pred, xs):
    return [x for x in xs if pred(x)]
```

2) На языке Scheme напишите функцию drop, принимающую список и целое число n и возвращающую исходный список без n первых элементов, например, так: `(drop '(1 2 3 4) 2) ⇒ (3 4)`

Решение:

```
(define (drop xs n)
    (if (or (zero? n) (null? xs))
        xs
        (drop (cdr xs) (- n 1))))
```

3) В программе определена функция:

```
def f(xs):
    ys = []
    for x in xs:
```

```
        if x not in ys:
            ys += [x]
    return ys
```

Определите результат вызова этой функции: f([2, 4, 2, 4])

Каково назначение этой функции?

Ответ:

1. [2, 4]
2. Эта функция возвращает новый список, составленный из элементов исходного так, что они не повторяются.

Об этом конспекте

Этот конспект - результат совместных бессонных стараний небольшой части группы ИУ9-11.

- Алена Ковинько
- Виктор Фадеев
- Давид Зухбая
- Саша Кручинина
- Сергей Максимов
- Ханага Агазаде

Версия от 18 января 2016 года 22:08 MSK