Билет 1

Экзаменационный билет № 1
по курсу «Основы информатики»

1. Понятие данных.
2. Модульное тестирование (юнит-тестирование), разработка через тестирование.
3. Задача

Задача № 1
к экзаменационному билету
по курсу «Основы информатики»

На языке Scheme запишите определение предиката, проверяющего, является ли аргумент символом (литерой) заглавной буквы латинского алфавита. *Не используйте* встроенные предикаты классификации символов.

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

   Источник < https://bmstu-iu9.github.io/scheme-labs/lect01.html>

2. **Модульное тестирование (юнит-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.

   Источник < https://bmstu-iu9.github.io/scheme-labs/lect08.html>

1

3. **Задача**

```
(define (my-char-upper-case? a)
  (and (<= (char->integer #\A) (char->integer a))
       (<= (char->integer a) (char->integer #\Z))))

(my-char-upper-case? #\S) -> #t
```

Билет 2

Экзаменационный билет №2
по курсу «Основы информатики»

1. Понятия программы, алгоритма.
2. Способы разбора и вычисления значения арифметического выражения, записанного в инфиксной нотации, с учетом приоритетов операций и скобок.
3. Задача

Задача №2
к экзаменационному билету
по курсу «Основы информатики»

На языке Python (Javascript) запишите определение предиката, проверяющего, является ли аргумент символом (литерой) заглавной буквы латинского алфавита. *Не используйте* встроенные предикаты классификации символов.

1. Понятия программы, алгоритма

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

Свойства алгоритма:

a. Дискретность — наличие структуры, разбитие на отдельные команды, понятия, действия.
b. Детерминированность — для одного и того же набора данных всегда один и тот же результат.
c. Понятность — элементы алгоритма должны быть понятны исполнителю.
d. Завершаемость — алгоритм завершается за конечное число шагов.
e. Массовость — применимость алгоритма для некоторого класса похожих задач.
f. Результативность — алгоритм должен выдавать результат.

Компьютерная программа — алгоритм, записанный на некотором языке программирования.

Источник < https://bmstu-iu9.github.io/scheme-labs/lect01.html>

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

Разбор выражения осуществляется с помощью лексических и синтаксических анализаторов.

В лексическом анализе текущее выражение разбивается на *лексемы,* они обрабатываются, и формируется последовательность *токенов*

Недопустимые символы отбрасываются.

На этапе синтаксического анализа формируется дерево разбора согласно грамматике:

```
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 .
```

Источник < https://bmstu-iu9.github.io/scheme-labs/home6.html>

Разбор осуществляется методом рекурсивного спуска.

Далее строится выражение (преобразовывается дерево разбора), вычисляемое с помощью конкретного языка программирования.

3. Задача (Python3)

```
def IsApper (a):
    if (a > 'A') and (a < 'Z'):
        return True
    else:
        return False

print(IsApper('S')) -> True
print(IsApper('s')) -> False
```

Билет 3

Экзаменационный билет №3
по курсу «Основы информатики»

1. Понятия типа данных, системы типов языка программирования, типизации.
2. Основные постулаты языков программирования семейства Lisp.
3. Задача

Задача №3
к экзаменационному билету
по курсу «Основы информатики»

На языке Scheme запишите определение процедуры, «удаляющей» из списка повторяющиеся элементы, например:

```
(remove-repeats '(1 1 2 4 2)) ⇒ (1 2 4)
```

1. Понятия типа данных, системы типов языка программирования, типизации.

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

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

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

Классификации систем типов:

i. Наличие системы типов: есть/нет.
ii. Типизация статическая/динамическая.
iii. Типизация явная/неявная.
iv. Типизация сильная/слабая.

Статическая и динамическая типизация:

- Статическая типизация — у каждой именованной сущности (переменной, функции…) есть свой фиксированный тип, он не меняется в процессе выполнения программы. Примеры: Си, C++, Java, Haskell, Rust, Go.
- Динамическая — тип переменной/функции известен только во время выполнения программы. Примеры: Scheme, JavaScript, Python.

Явная и неявная типизация:

- Явная — тип данных для сущностей явно записывается в программе. Например, int x в языке Си. Языки с явной типизацией: Си, C++, Java и т.д.
- Неявная — тип данных можно не указывать. Неявная типизация характерна прежде всего для динамически типизированных языков. В статически типизированных языках используется совместно с выводом типов. Вывод типов переменных присутствует в следующих языках: C++ (ключевое слово auto), Go (когда тип переменной не указан), Rust, Haskell и т.д.

Типизация сильная/слабая:

- Сильная — неявные преобразования типов запрещены. Например, нельзя сложить строку и число. Языки с сильной типизацией: Scheme, Python, Haskell.
- Слабая типизация — неявные преобразования допустимы. Например, в JavaScript при сложении строки с числом число преобразуется в строку. Если в JavaScript в переменной лежит строка с последовательностью цифр, то, при умножении её на число, она неявно преобразуется в число: '1000' * 5 → 5000. Примеры языков: JavaScript, Си, Perl, PHP.

Источник < https://bmstu-iu9.github.io/scheme-labs/lect06b.html>

2. Основные постулаты языков программирования семейства Lisp.

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

Источник < https://bmstu-iu9.github.io/scheme-labs/lect02.html>

3. Задача

```
(define (remove-repeats xs)
  (if (null? xs)
      '()
      (if (member (car xs) (cdr xs))
          (remove-repeats (cdr xs))
          (cons (car xs) (remove-repeats (cdr xs))))))

(remove-repeats '(1 1 2 4 4 5 4 2 3 6 2 8))
```

Билет 4

Экзаменационный билет № 4
по курсу «Основы информатики»

1. Важнейший парадигмы программирования и их отличительные черты.
2. Общая характеристика языка Scheme.
3. Задача

Задача № 4
к экзаменационному билету
по курсу «Основы информатики»

На языке Python (Javascript) запишите определение функции, «удаляющей» из списка повторяющиеся элементы, например:

```
remove_repeats([1, 1, 2, 4, 2]) ⇒ [1, 2, 4]
```

1. Важнейшие 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 делятся на три большие группы:

a. Императивное программирование.
b. Декларативное программирование.
c. Метапрограммирование.

Императивное программирование — способ записи программ, в котором указывается последовательность действий.

Основной признак императивной парадигмы (группы парадигм) — оператор деструктивного присваивания. Слово «деструктивное» означает, что присваивание может изменять значение, хранящееся в переменной — старое теряется безвозвратно, заменяясь новым значением.

Декларативное программирование — способ записи программ, в котором описываются взаимосвязь между данными; описывается цель, а не последовательность шагов для её достижения. Деструктивного присваивания в декларативной парадигме нет. Возможно лишь однократное присваивание значения при создании новой переменной.

Декларативная парадигма:

a. Функциональное программирование — алгоритм описывается как набор функций; порядок вычисления функций не существенен и на результат влиять не должен. В ленивых языках (например, Haskell) функции вызываются только когда нужен их результат.
b. Логическое программирование — алгоритм описывает взаимосвязь между понятиями; выполнение программы сводится к выполнению запросов. Представлено почти исключительно языком Prolog, сильно отчасти — SQL.

Метапрограммирование — программа становится объектом управления со стороны программы — той же или другой.

Парадигма метапрограммирования:

a. Программы пишут программы: макросы, генераторы кода, шаблонное метапрограммирование в C++.
b. Рефлексия (интроспекция) — программы взаимодействуют с вычислительной средой.

Источник < https://bmstu-iu9.github.io/scheme-labs/lect01.html>

2. Общая характеристика языка Scheme.

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

Сохраняет основные постулаты языков семейства Lisp:

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

Всё записывается в виде списка, его грамматику можно записать как БНФ:

```
<терм>   ::= <атом> | <список>
<список> ::= (<термы>)
<термы>  ::= <пусто> | <терм> <термы>
<атом>   ::= <переменная> | <число> | <символ> | <строка>
```

В выражениях языка Scheme после открывающей круглой скобки указывается операция. Операцией может быть либо вызов функции, либо так называемая особая форма. В случае вызова функции первым термом после скобок является имя функции или выражение, порождающее функцию. В случае особой формы после открывающей круглой скобки располагается ключевое слово. (*Можно написать сильно больше, читать источник*)

Источник < https://bmstu-iu9.github.io/scheme-labs/lect02.html>

3. Задача (Python)

```python
def remove_repeats(xs):
    res = []
    for x in xs:
        if x not in res:
            res.append(x)
    return res

xs = [1, 2, 2, 1, 3, 4, 4, 5, 7, 5]
xs = remove_repeats(xs)
print(xs)
```

Билет 5

Экзаменационный билет №5
по курсу «Основы информатики»

1. Каким образом в язык программирования с динамической типизацией можно ввести новый тип данных? Приведите примеры.
2. Типизация и система типов языка Scheme.
3. Задача

Задача №5
к экзаменационному билету
по курсу «Основы информатики»

На языке Scheme запишите определение процедуры `filter`, принимающей предикат одного аргумента и список и возвращающей список из тех элементов исходного списка, которые удовлетворяют предикату.

1. Каким образом в язык программирования с динамической типизацией можно ввести новый тип данных? Приведите примеры.

Динамическая типизация -
тип переменной/функции известен только во время выполнения программы. Примеры: Scheme, JavaScript, Python.

В Python, JavaScript и многих других есть встроенные языковые средства для создания своих типов данных (Class, Structure)

В Scheme таких средств нет, поэтому всё проектирование осуществляется вручную с помощью процедур и макросов.

Для типов данных языка Scheme обычно определены четыре вида операций:

- конструктор — процедура, имя которой имеет вид make-‹имя-типа›, например, make-vector, make-set (см. дз), конструктор предназначен для создания новых значений данного типа,
- предикат типа — процедура, возвращающая #t, если её аргумент является значением данного типа, имеет имя ‹имя-типа›?: vector?, set? (см. дз), multi-vector? (см. дз),
- модификаторы — операции, меняющие на месте содержимое объекта, их имя имеет вид ‹тип›-‹операция›!, например, vector-set!, multivector-set! (см. дз),
- прочие операции имеют имя вида ‹тип›-‹операция›, vector-ref, set-union, string-append и т.д.

Пользовательские типы данных часто представляют как списки, первым элементом которых является символ с именем типа, а остальные — хранимые значения.

Пример. Тип данных — круг.

```
(define (make-circle x y r)
  (list 'cicrle x y r))
(define (circle? c)
  (and (list? c) (equal? (car c) 'circle)))
(define (circle-center c)
  (list (cadr c) (caddr c)))      ; (cadr xs) = (car (cdr xs))
(define (circle-radius c)
  (cadddr c))
(define (circle-set-center! c p)
  (let ((x (car p))
        (y (cadr p)))
    (set-car! (cdr c) x)
    (set-car! (cddr c) y)
    c))
(define (circle-set-radius! c r)
  (set-car! (cdddr c) r)
  c)
```

Источник < https://bmstu-iu9.github.io/scheme-labs/lect06b.html>

2. Типизация и система типов языка Scheme

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

Классификации систем типов:

i. Наличие системы типов: есть/нет.
ii. Типизация статическая/динамическая.
iii. Типизация явная/неявная.
iv. Типизация сильная/слабая.

Статическая и динамическая типизация:

- Статическая типизация — у каждой именованной сущности (переменной, функции…) есть свой фиксированный тип, он не меняется в процессе выполнения программы. Примеры: Си, C++, Java, Haskell, Rust, Go.
- Динамическая — тип переменной/функции известен только во время выполнения программы. Примеры: Scheme, JavaScript, Python.

Явная и неявная типизация:

- Явная — тип данных для сущностей явно записывается в программе. Например, int x в языке Си. Языки с явной типизацией: Си, C++, Java и т.д.
- Неявная — тип данных можно не указывать. Неявная типизация характерна прежде всего для динамически типизированных языков. В статически типизированных языках используется совместно с выводом типов. Вывод типов переменных присутствует в следующих языках: C++ (ключевое слово auto), Go (когда тип переменной не указан), Rust, Haskell и т.д.

Типизация сильная/слабая:

- Сильная — неявные преобразования типов запрещены. Например, нельзя сложить строку и число. Языки с сильной типизацией: Scheme, Python, Haskell.
- Слабая типизация — неявные преобразования допустимы. Например, в JavaScript при сложении строки с числом число преобразуется в строку. Если в JavaScript в переменной лежит строка с последовательностью цифр, то, при умножении её на число, она неявно преобразуется в число: '1000' * 5 → 5000. Примеры языков: JavaScript, Си, Perl, PHP.

3. Задача

```
(define (filter pr? xs)
  (if (null? xs)
      '()
      (if (pr? (car xs))
          (cons (car xs) (filter pr? (cdr xs)))
          (filter pr? (cdr xs)))))

(filter odd? '(1 2 3 4 5 6 7 8))
```

Билет 6

Экзаменационный билет №6
по курсу «Основы информатики»

1. Понятие абстрактного типа данных. Примеры.
2. Простые типы языка Scheme и основные операции над ними.
3. Задача

Задача №6
к экзаменационному билету
по курсу «Основы информатики»

На языке Python (Javascript) запишите свое собственное определение встроенной функции `filter`, принимающей предикат одного аргумента и список и возвращающей список из тех элементов исходного списка, которые удовлетворяют предикату.

1. Понятие абстрактного типа данных. Примеры

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

Примеры: структуры и классы в языках c++ и Python, типы данных, описанные в языке Scheme (см. Билет 5)

Источник <https://bmstu-iu9.github.io/scheme-labs/lect06b.html>

2. Простые типы языка Scheme и основные операции над ними.

1) *Числа: целые, дробные, вещественные, комплексные (1, 3.14, 3/4, 10+7.5i)*

Операции: все арифметические (*, /, +, expr…)

Предикаты определения типа:

```
(number? x)          ; это число
(complex? x)         ; комплексное число
(real? x)            ; вещественное число
(rational? x)        ; дробное число
(integer? x)         ; целое число
(exact? num)
(inexcat? num)
```

Операции преобразования типов:

```
(exact->inexact num)    → ближайшее вещественное число
(inexact->exact num)    → ближайшее дробное число
```

Источник <https://bmstu-iu9.github.io/scheme-labs/lect07.html>

2) *Литерный тип (character) (#\a, #\tab, #\space…)*

Операции:

Преобразование между символом и его числовым кодом:

```
(char->integer char)      → code
(integer->char code)      → char
(char->integer #\@)       → 48
(integer->char 48)        → #\@
```

Сравнение символов (по числовым кодам):

```
(char<? ch1 ch2)
(char>? ch1 ch2)
(char<=? ch1 ch2)
(char>=? ch1 ch2)
(char=? ch1 ch2)
```

Сравнение без учёта регистра:

```
(char-ci<? ch1 ch2)
(char-ci>? ch1 ch2)
...
```

Преобразование регистра (??):

```
(char-upcase #\a)         → #\A
(char-downcase #\Q)       → #\q
(char-upcase #\1)         → #\1
(char-downcase #\!)       → #\!
```

Предикаты видов литер:

```
(char-whitespace? ch)   ; пробельный символ: пробел, табуляция,
                        ; перевод строки и т.д.
(char-numeric? ch)      ; цифра
(char-alphabetic? ch)   ; буква
(char-upper-case? ch)   ; большая буква
(char-lower-case? ch)   ; строчная буква
```

Источник <https://bmstu-iu9.github.io/scheme-labs/lect07.html>

3) *Логический тип (#t, #f)*

4) *cons-ячейка (list)*

(car xs) - первый элемент

(cdr xs) - все кроме первого, *иначе* - второй элемент cons ячейки

(pair? xs) - предикат проверки

3. Задача (Python)

```python
def is_odd(n):
    return n % 2 == 1

def filter(predicate, xs):
    res = []
    for x in xs:
        if predicate(x):
            res.append(x)
    return res

print(filter(is_odd, [1, 2, 3, 4, 5, 6, 7, 8]))
```

Билет 7

Экзаменационный билет № 7
по курсу «Основы информатики»

1. Функции (процедуры) высшего порядка в языках программирования высокого уровня.
2. Составные типы языка Scheme и основные операции над ними.
3. Задача

Задача № 7
к экзаменационному билету
по курсу «Основы информатики»

На языке Scheme запишите свое собственное определение встроенной процедуры `map`:

а) Принимающей процедуру строго одного аргумента и строго один список значений.

б) Принимающей процедуру произвольного числа аргументов и соответствующее число списков значений.

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

В Scheme - конструкция lambda создаёт безымянную процедуру вида

(lambda (аргументы) выражение)

При вызове процедуры создаются новые переменные, соответствующие формальным параметрам и они связываются с фактическими параметрами.

```
(define f
  (lambda (x y) (+ x y)))
(f 10 13)
```

Можно передать в процедуру в виде параметра:

```
(define (g f)
  (f 10 13))
(g (lambda (x y) (+ x y)))
(g +)
```

Источник < https://bmstu-iu9.github.io/scheme-labs/lect03.html>

В скриптовых языках также есть похожие конструкции, определяющие процедуры высших порядков. Например, в Python:

```
lambda x: x ** 2 - безымянная функция, возводящая число в квадрат.
```

Функция filter принимает функцию предикат и итератор, возвращает итератор, элементами которого являются данные из исходного итератора, для которых предикат возвращает True:

Пример:

```
>>> list(filter(lambda x: x > 0, [-1, 1, -2, 2, 0]))
[1, 2]
```

Функция map принимает функцию и итератор, возвращает итератор, элементами которого являются результаты применения функции к элементам входного итератора.
Пример:

```
a = [1, 2, 3, 4, 5]
>>> list(map(lambda x: x**2, a))
[1, 4, 9, 16, 25]
```

2. Составные типы языка Scheme и основные операции над ними.

1) Строковый тип (String) - содержит только элементы типа char

Примеры:

```
"Hello!"
"First line\nSecond line"
```

Создание строк:

```
(make-string count char)
(make-string count)
```

Присваивание:

```
(string-ref str k)            → k-й символ строки
(string-set! str k char)      ; присваивает k-му символу
```

Сравнение строк (в лексикографическом порядке), с учётом регистра и без него (с суффиксом -ci):

```
(string=? str1 str2)
(string<? str1 str2)
...
(string-ci=? str1 str2)
(string-ci<? str1 str2)
```

Длина строки:

```
(string-length "abcdef")      → 6
```

Выбор подстроки:

```
(substring "abcdef" 2 5)      → "cde"
(substring "abcdef" 2 3)      → "c"
```

Преобразование типа: (list->string xs)
Предикат: (string? xs) -> #t|#f

Источник < https://bmstu-iu9.github.io/scheme-labs/lect07.html>

2) Список (list)
Создание: (list 1 2 3..)
Операции:
(list-ref xs x) - аналогично string
(car xs) - первый элемент
(cdr xs) - все, кроме первого
(грамотнее рассуждать с точки зрения cons-ячейки, как в билете 6)
Операции преобразования типов:
(vector->list vec)
(string->list str)
Предикат: (list? xs)

3) *Vector*
Поддерживаются все операции, что и со string.
Вызов принимает вид (vector-<func> vec)

Пример: (vector-ref vec k) - вернет k-тый элемент вектора vec.

Преобразование типа: (list->vector xs)
Предикат: (vector? xs) -> #t|#f

3. Задача

```
;a)
(define (my-map1 func xs)
  (if (null? xs)
      '()
      (cons (func (car xs)) (my-map1 func (cdr xs)))))

(my-map1 (lambda (x) (+ x 1)) '(1 2 3 4))
;b)
(define (my-map2 func . xss)
  (letrec ((len (length xss))
           (len-arg (length (car xss)))
           (take-cars (lambda (xss1 i n xs)
                        (if (= i n)
                            xs
                            (take-cars xss1 (+ 1 i) n (append xs (list (car (list-ref xss1 i))))))))
           (cdrs (lambda (xss1 i n xss2)
                   (if (= i n)
                       xss2
                       (cdrs xss1 (+ i 1) n (append xss2 (list (cdr (list-ref xss1 i))))))))
           (loop (lambda (func xss1 xs)
                   (if (null? (car xss1))
                       xs
                       (loop func (cdrs xss1 0 len '()) (append xs (list (apply func (take-cars xss1 0 len '()))))))))
    (loop func xss '())))
(my-map2 + '(1 2 3 4) '(5 6 7 8) '(9 0 2 3))
```

Билет 8

Экзаменационный билет №8
по курсу «Основы информатики»

1. Способы организации повторяющихся вычислений в языках программирования высокого уровня.
2. Символьный тип в языке Scheme и его применение.
3. Задача

Задача №8
к экзаменационному билету
по курсу «Основы информатики»

На языке Python (Javascript) запишите свое собственное определение встроенной функции `map`:

а) Принимающей процедуру строго одного аргумента и строго один список значений,

б) Принимающей процедуру произвольного числа аргументов и соответствующее число списков значений.

1. Способы организации повторяющихся вычислений в языках программирования высокого уровня.

В большинстве языков используется рекурсия и циклы, везде имеют похожий функционал.
В Scheme вместо циклов часто используется хвостовая рекурсия, которая не отличается от цикла по вычислительной сложности.

2. Символьный тип в языке Scheme и его применение.

Символьный тип данных — это «зацитированное», «замороженное» имя:
Имеет известный нам предикат типа и функции преобразования:

```
(symbol? 'hello)                → #t
(symbol? "hello")               → #f
(symbol? #\a)                   → #f
(symbol->string 'hello)         → "hello"
(string->symbol "hello")        → hello
(string->symbol
  (string-append
    (symbol->string 'hell)
    (symbol->string 'o))))      → hello
```

Применяется в основном для превращения программы в данные, и последующего их исполнения.
Источник < https://bmstu-iu9.github.io/scheme-labs/lect08.html>

3. Задача

```python
def func(n):
    return n ** 2

def my_map1(function, s):
    for i in range(len(s)):
        s[i] = function(s[i])
    return s
def multi_plus(*args):
    res = 0
    for x in args:
        res += x
    return res

def my_map2(function, *args):
    def cars(xss):
        list_cars = []
        for x in xss:
            list_cars.append(x[0])
        return list_cars
    def cdrs(xss):
        list_cdrs = []
        for x in xss:
            list_cdrs.append(x[1:])
        return list_cdrs
    res = []
    def loop(xss):
        if len(xss[0]) == 0:
            return
        else:
            res.append(function(*(cars(xss))))
            loop(cdrs(xss))
    loop(args)
    return res


print(my_map1(func, [1, 2, 3, 4, 5]))
print(my_map2(multi_plus, [1, 2, 3], [3, 7, 1], [4, 7, 6]))
```

Билет 9

Экзаменационный билет №9
по курсу «Основы информатики»

1. Мемоизация результатов вычислений.
2. Применение процедур (функций) высшего порядка для обработки списков на языке Scheme.
3. Задача

Задача №9
к экзаменационному билету
по курсу «Основы информатики»

На языке Python (Javascript) напишите собственное определение функции **reduce**. Через эту функцию определите специальные виды свертки: сумму, произведение, нахождение минимального элемента списка и максимального элемента списка.

1. Мемоизация результатов вычислений.

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

Пример:

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))))))
```

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

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

Примеры: map - применяет переданную функцию к каждому элементу переданного списка
apply - раскрывает переданный список и передает в переданную функцию

Применяются данные функции как раз в случаях, когда надо применить функцию к большому количеству аргументов, при этом избегая написания циклов, хвостовых рекурсий и т.д.

3. Задача (Python)

```
def my_sum(a, b):
    return a + b

def pr(a, b):
    return a * b

def my_reduce (function, xs):
    if (len(xs) == 1):
        return xs[0]
    else:
        return function(xs[0], my_reduce(function, xs[1:]))

print(my_reduce(my_sum, [1, 2, 3, 4, 5]))
print(my_reduce(pr, [1, 2, 3, 4, 5]))
print(my_reduce(min, [1, 2, 3, 4, 5]))
print(my_reduce(max, [1, 2, 3, 4, 5]))
print(my_reduce(lambda x, y: x ** y, [5, 3, 2]))
```

Билет 10

Экзаменационный билет № 10
по курсу «Основы информатики»

1. Нестрогие и отложенные вычисления. Примеры.
2. 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).
3. Задача

Задача № 10
к экзаменационному билету
по курсу «Основы информатики»

На языке Scheme запишите собственные определения встроенных процедур:

а) `list-ref`,

б) `length`,

в) `reverse`.

1. Нестрогие и отложенные вычисления. Примеры.

Нестрогие вычисления означают, что аргументы не вычисляются до тех пор, пока их значение не используется в теле функции. В случае нестрогих вычислений значения выражений могут вычисляться по необходимости, их вызов может быть отложен.

Примеры:

- or и and в языке Scheme
- `(if cond then else)` — вычисляется либо `then`, либо `else`
- в языке Си логические операции &&, || тоже не строгие

Отложенные вычисления (ленивые)- вычисления откладываются до тех пор, пока не понадобится их результат. Можно организовать с помощью delay и force в Scheme и функций-генераторов и ключевого слова yield в Python.

Пример:

язык программирования Хаскель

```
test xs = head (map (\x -> x*x) xs)
```

Будет вычисляться квадрат только самого первого элемента списка.

Источник - < https://bmstu-iu9.github.io/scheme-labs/lect09b.html>

2. 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 ...)))))
```

Задача 3

```
(define (my-list-ref xs i)
  (if (> i (- (length xs) 1))
      #f
      (if (eq? i 0)
          (car xs)
          (my-list-ref (cdr xs) (- i 1)))))

(my-list-ref '(1 2 3 4) 3)

(define (my-length xs)
  (if (null? xs)
      0
      (+ 1 (my-length (cdr xs)))))

(my-length '(1 2 3 4 5 6 7))

(define (my-reverse xs)
  (letrec ((loop (lambda (xs1 xs2)
                   (if (null? xs1)
                       xs2
                       (loop (cdr xs1)
                             (append (list (car xs1)) xs2))))))
          (loop xs '())))

(my-reverse '(1 2 3 4 5))
```

Билет 11

Экзаменационный билет № 11
по курсу «Основы информатики»

1. Способы реализации языка программирования высокого уровня.
2. Лексические замыкания (на примере) в языке Scheme. Свободные и связанные переменные. Использование лексического замыкания для определения процедуры со статической переменной.
3. Задача

Задача № 11
к экзаменационному билету
по курсу «Основы информатики»

На языке Python (Javascript) запишите определение функции, осуществляющей подсчет элементов списка, удовлетворяющих предикату.

1. Способы реализации языка программирования высокого уровня.
Языки программирования могут быть реализованы как компилируемые или интерпретируемые.

Транслятор - программа, предназначенная для перевода программы, написанной на одном языке программирования, в программу на другом языке программирования. Процесс перевода называется *трансляцией*. Тексты исходной и результирующей программ находятся в памяти компьютера. Примером транслятора является компилятор.

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

Существует другой способ сочетания процессов трансляции и выполнения программы. Он называется *интерпретацией*. Суть процесса интерпретации состоит в следующем. Вначале переводится в машинные коды, а затем выполняется первая строка программы. Когда выполнение первой строки окончено, начинается перевод второй строки, которая затем выполняется и так далее. Управляет этим процессом программа-интерпретатор.

Интерпретатор- программа, выполняющая интерпретацию. Интерпретация - пооператорный (покомандный) анализ, обработка и тут же выполнение программы или запроса.
Источник - < https://vuzlit.ru/953214/sposoby_realizatsii_yazykov_programmirovaniya>

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

*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))
```

3. Задача (Python)

```
def is_even(x):
    return not(x % 2)

def count_pr(pr, xs):
    res = 0
    for x in xs:
        if (pr(x)):
            res+=1
    return res

print(count_pr(is_even, [1, 2, 3, 4, 5, 6]))
```

Билет 12

Экзаменационный билет № 12
по курсу «Основы информатики»

1. Компилятор и интерпретатор: определение, основные функциональные элементы.
2. Особенности логических операций в языке программирования Scheme.
3. Задача

Задача № 12
к экзаменационному билету
по курсу «Основы информатики»

На языке Scheme запишите определение процедуры, осуществляющей подсчет элементов списка, удовлетворяющих предикату.

1. 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. Генерация промежуточного представления
5. Оптимизация (необязательный этап) удаление излишних конструкций и упрощение кода с сохранением его смысла.
6. Генерация кода машинного кода или байткода. (Байткод - промежуточное представление, в которое может быть переведена компьютерная программа. Это компактное представление программы, уже прошедшей синтаксический и семантический анализ.)
7. Компоновка

Интерпретатор- программа, выполняющая интерпретацию. Интерпретация - пооператорный (покомандный) анализ, обработка и тут же выполнение программы или запроса.
Стадии интерпретациии:
1. Лексический анализ
2. Синтаксический анализ
3. Семантический анализ
4. Исполнение

Источник - < https://linksharing.samsungcloud.com/wbGChqgOlCC4>

2. Особенности логических операций в языке программирования Sheme.
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.

```
(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)
```

3. Задача

```
(define (count_pr pr? xs)
  (if (null? xs)
      0
      (if (pr? (car xs))
          (+ 1 (count_pr pr? (cdr xs)))
          (count_pr pr? (cdr xs)))))

(count_pr even? '(1 2 3 4 5 6 7 8))
```

Билет 13

Экзаменационный билет № 13
по курсу «Основы информатики»

1. Лексический анализатор: назначение, входные данные, выходные данные, принцип реализации.
2. Гигиенические макросы в языке Scheme.
3. Задача

Задача № 13
к экзаменационному билету
по курсу «Основы информатики»

На языке Scheme напишите собственные определения специальных видов свертки: суммы, произведения, нахождения минимального элемента списка и максимального элемента списка.

1. 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.).

2. Гигиенические макросы в языке Scheme.

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

<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:

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

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

3. Задача

```
(define (list-sum xs)
  (if (null? xs)
      0
      (+ (car xs) (list-sum (cdr xs)))))

(list-sum '(1 2 3 4 5))

(define (list-mul xs)
  (if (null? xs)
      1
      (* (car xs) (list-mul (cdr xs)))))

(list-mul '(1 2 3 4 5))

(define MAX_ELEM 1e9)
(define MIN_ELEM -1e9)

(define (list-min xs)
  (if (null? xs)
      MAX_ELEM
      (min (car xs) (list-min (cdr xs)))))

(list-min '(3 1 3 5 7))

(define (list-max xs)
  (if (null? xs)
      MIN_ELEM
      (max (car xs) (list-max (cdr xs)))))

(list-max '(3 1 3 5 7))
```

Билет 14

Экзаменационный билет № 14
по курсу «Основы информатики»

1. Синтаксический анализатор: назначение, входные данные, выходные данные.
2. Продолжения в языке Scheme.
3. Задача

Задача № 14
к экзаменационному билету
по курсу «Основы информатики»

В некоторой программе на языке Python (Javascript) множества представлены в виде списков (Javascript: в виде массивов), элементы которых не повторяются. Используя списковые включения (list comprehensions), запишите определения операций над множествами (Javascript: используйте функции высших порядков для обработки последовательностей):

а) разности,

б) симметрической разности,

в) декартового произведения.

1. 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.

2. 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.

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.

3. Задача (Python)

```
def difference(xs1, xs2):
    res = [elem for elem in xs1 if elem not in xs2]
    return res

def sim_difference(xs1, xs2):
    return difference(xs1, xs2) + difference(xs2, xs1)

def decart(xs1, xs2):
    res = [(a, b) for a in xs1 for b in xs2]
    return res


print(difference([1, 2, 3, 4, 5], [2, 3]))
print(sim_difference([1, 2, 3, 4, 5], [2, 3, 6, 7]))
print(decart([1, 2, 3], [4, 5, 6]))
```

Билет 15

Экзаменационный билет № 15
по курсу «Основы информатики»

1. Формальная грамматика, терминальные символы, нетерминальные символы.
2. Ввод-вывод в языке Scheme.
3. Задача

Задача № 15
к экзаменационному билету
по курсу «Основы информатики»

На языке Scheme запишите определение процедуры, которая принимает выражение вида $a * x + a * y$, записанное на языке Scheme в префиксной нотации (a — целое число, а x и y — переменные или выражения) и возвращает выражение вида $a \cdot (x + y)$, также записанное на языке Scheme в инфиксной нотации и пригодное для вычисления его значения с помощью встроенной процедуры `eval` в окружении, в котором определены соответствующие символы. В случае, если преобразование не может быть выполнена, процедура должна возвращать `#f`. Примеры применения процедуры:

```
(recast '(+ (* 2 x) (* 2 y))) ⇒ (* 2 (+ x y))
(recast '(+ (* 2 (- a b)) (* 2 (- p q)))) ⇒ (* 2 (+ (- a b) (- p q)))
(recast '(expt x 2)) ⇒ #f
```

1. 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.

Источник - < https://linksharing.samsungcloud.com/wbGChqgOlCC4>

2. 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
- (writechar 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
- (readchar port) считывание одного символа из указанного порта
- (peekchar port) считывание следующего символа из порта, при этом переход на следующий символ не осуществляется

3. Задача

```
(define (recast xs)
  (if (and (= (length xs) 3) (list? (list-ref xs 1)) (list? (list-ref xs 2)) (= (length (list-ref xs 1)) 3) (= (length (list-ref xs 2)) 3))
      (let ((sign (list-ref xs 0))
            (sign1 (list-ref (list-ref xs 1) 0))
            (mul1 (list-ref (list-ref xs 1) 1))
            (sign2 (list-ref (list-ref xs 2) 0))
            (mul2 (list-ref (list-ref xs 2) 1))
            (op1 (list-ref (list-ref xs 1) 2))
            (op2 (list-ref (list-ref xs 2) 2)))
        (if (and (eq? sign '+) (eq? mul1 mul2) (eq? sign1 sign2) (eq? sign1 '*))
            `(* ,mul1 (+ ,op1 ,op2))
            #f))
      #f))

(recast '(+ (* 2 x) (* 2 y)))
(recast '(+ (* 2 (- a b)) (* 2 (- p q))))
(recast '(expt x 2))
```

Билет 16

Экзаменационный билет №16
по курсу «Основы информатики»

1. БНФ.
2. Средства для метапрограммирования языка Scheme.
3. Задача

2. Средства для метапрограммирования языка Scheme

Средства метапрограммирования («код как данные», макросы)

Макрос — это инструмент переписывания кода. Т.е. способ создать новую языковую конструкцию на основании имеющихся.

Процесс выполнения выражений на Scheme. Пусть у нас есть выражение вида (‹имя› ‹термы…›)

I. Если ‹имя› — ключевое слово языка (if, define, quote, lambda, и т.д.), то выражение интерпретируется как особая форма.
II. Если ‹имя› — имя макроса, то данное выражение перезаписывается согласно определению макроса.
III. Если ‹имя› — имя переменной, то в переменной должна быть процедура, эта процедура вызывается.

Т.е. можно считать, что вычисление выражения состоит из двух этапов:

I. Раскрытие макросов.
II. Собственно вычисления (выполнения особых форм, вызовы процедур).

Синтаксис определения макроса:

```
(define-syntax ‹имя›
       (syntax-rules (‹ключевые слова›)
         (‹образец› ‹шаблон›)
         (‹образец› ‹шаблон›)
         (‹pattern› ‹template›)))
```

‹образец› (‹pattern›) — вид, который должно иметь обращение к макросу.
‹шаблон› (‹template›) — то, на что макрос заменяется.

Образцы проверяются сверху вниз и выбирается тот, который первым подходит.

В правилах макроса могут быть переменные (правильнее сказать, метапеременные), которым соответствуют фрагменты кода на Scheme. Если в ‹образце› мы можем вместо вхождений переменных подставить фрагменты кода на Scheme таким образом, что получим запись применения макроса, то считаем, что применение макроса с образцом сопоставилось успешно, и правило применяется.

Источник < https://bmstu-iu9.github.io/scheme-labs/lect08.html>

Задача №16
к экзаменационному билету
по курсу «Основы информатики»

Дано равенство вида: $x_1 \star x_2 \star x_3 = x_4 \star x_5 \star x_6$, где $x_1 \dots x_6$ — целые числа в диапазоне от 1 до 100, а $\star$ – операция сложения, деления, вычитания или деления нацело. В одном равенстве могут встречаться как одинаковые, так и разные числа и операции. На языке Python напишите определение функции, которая возвращает все возможные равенства в виде списка строк для каждого переданного ей сочетания чисел слева и справа от знака равенства.

Пример вызова:

```
find([63, 9, 28], [7, 4, 7])
```

Пример результата:

```
63//9*28 = 7*4*7
63//9+28 = 7*4+7
63//9+28 = 7+4*7
63//9-28 = 7-4*7
63//9//28 = 7//4//7
```

1. БНФ

Форма Бекуса-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> )
```

При описании многих языков программирования (в учебниках, стандартах) используется тот или иной вариант БНФ. Нотация может быть расширена такими обозначениями как * или + после нетерминала, означающие повторение ноль или более раз (*) или один или более раз (+) данного нетерминала.

Как правило, если записана грамматика языка программирования, то под аксиомой подразумевается самый первый нетерминал (в примере <Выражение>).

Источник < https://bmstu-iu9.github.io/scheme-labs/lect13.html>

3. Задача

```
op_list = {'**': lambda x, y, z: x * y * z, '*-': lambda x, y, z: x * y - z,
           '*+': lambda x, y, z: x * y + z, '*//': lambda x, y, z: x * y // z,
           '//*': lambda x, y, z: x // y * z, '//-': lambda x, y, z: x // y - z,
           '//+': lambda x, y, z: x // y + z, '////': lambda x, y, z: x // y // z,
           '+*': lambda x, y, z: x + y * z, '+-': lambda x, y, z: x + y - z,
           '++': lambda x, y, z: x + y + z, '+//': lambda x, y, z: x + y // z,
           '-*': lambda x, y, z: x - y * z, '--': lambda x, y, z: x - y - z,
           '-+': lambda x, y, z: x - y + z, '-//': lambda x, y, z: x - y // z}
def find(left_arg, right_arg):
    for op_1 in ("*", "+", "-", "//"):
        for op_2 in ("*", "+", "-", "//"):
            for op_3 in ("*", "+", "-", "//"):
                for op_4 in ("*", "+", "-", "//"):
                    if (op_list[op_1 + op_2](left_arg[0], left_arg[1], left_arg[2]) ==
                            op_list[op_3 + op_4](right_arg[0], right_arg[1], right_arg[2])):
                        print(left_arg[0], op_1, left_arg[1], op_2, left_arg[2], "=",
                              right_arg[0], op_3, right_arg[1], op_4, right_arg[2])

(find([63, 9, 28], [7, 4, 7]))
```

Билет 17

Экзаменационный билет №17
по курсу «Основы информатики»

1. $LL(1)$-грамматика: особенности и их использование.
2. Хвостовая рекурсия и ее оптимизация интерпретатором языка Scheme.
3. Задача

Задача №17
к экзаменационному билету
по курсу «Основы информатики»

На языке Scheme напишите определение процедуры для конкатенации символических имен (литеральных констант). Пример применения процедуры:

```
(symbols-append 'foo 'bar) ⇒ foobar
(symbols-append 'foo '- 'bar) ⇒ foo-bar
```

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

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.

Пример: не-LL(k)-грамматика:

E → T | E + T
T → F | T * F
F → n | (E)
Если имеем строку n * n * n + n + n и нетерминал E, то мы не знаем, по какому правилу нужно раскрывать E. Поскольку в начале строки может быть сколько угодно сомножителей, в общем случае, чтобы выбрать правило раскрытия для E (т.е. E → T или E → E + T), нужно прочитать неизвестное количество входных знаков. А для LL(k)-грамматики k должно быть конечно и фиксировано.

Пример: LL(1)-грамматика для тех же арифметических выражений:

E → T E′
E′ → ε | + T E′
T → F T′
T′ → ε | * F T′
F → n | (E)
здесь ε — пустая строка. В данной грамматике мы всегда можем определить применимое правило. Например, для E правило только одно, его используем. Для E′: если строка начинается на +, то выбираем вторую ветку E′ → + T E′, иначе выбираем первую E′ → ε. Для F: знак n выбирает первую ветку, знак (— вторую.

Пример не-LL(1)-грамматики:

A → B x z
B → ε | x y
Для строки x … и нетерминала B мы не можем определить раскрытие по первому символу, т.к. допустимо и то, и другое правило. Язык включает в себя две строки: x z и x y x z. По первому символу невозможно определить правило для B.

Однако, это грамматика LL(2). По первым двум символам определить раскрытие можно.

Также грамматика не LL(1) если разные правила начинаются с одинаковых символов:

A → x a | x b
Грамматика не LL(1), если в правилах имеем т.н. левую рекурсию:

A → x | A y
Преимущество LL(1)-грамматик — для них сравнительно легко написать синтаксический анализатор методом рекурсивного спуска.

Источник < https://bmstu-iu9.github.io/scheme-labs/lect13.html>

3. Задача

```
(define (symbols-append . xs)
  (define (list-symbol->string xs)
    (cond ((null? xs) "")
          (else (string-append (symbol->string (car xs))
                               (list-symbol->string (cdr xs))))))
  (string->symbol (list-symbol->string xs)))
(symbols-append 'foo 'bar) ;⇒ foobar
(symbols-append 'foo '- 'bar) ;⇒ foo-bar
```

2. Хвостовая рекурсия и ее оптимизация интерпретатором языка Scheme

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

```
(define (f x y z)
  (if (a)
      (b x (c y))
      (d (if (e)
             (g)
             (h)))))
```
(Вызовы b и d - хвостовые)

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 итерации по вычислительным затратам.

Рекурсивный факториал:

```
(define (fact N)
  (if (> N 0)
      (* (fact (- N 1)) N)
      1))
```
Итеративный факториал:

```
(define (fact N)
  (define (loop i res)
    (if (<= i N)
        (loop (+ i 1)
              (* res i))
        res))
  (loop 1 1))
```
Оптимизация хвостовой рекурсии изнутри:

```
int loop(int N, int i, int res) {
  if (i <= N) {
    loop(N, i + 1, res * i);
  } else {
    return res;
  }
}

int loop(int N, int i, int res) {
LOOP:
  if (i <= N) {
    res = res * i;
    i = i + 1;
    goto LOOP;
  } else {
    return res;
  }
}
```
Вольный перевод на Scheme: (не из лекции)
```
(define (loop n i res)
  (if (<= i n)
      (loop n (+ i 1) (* res i))
      res))
(define (fact n)
  (define (loop n i res)
    (do ((n n)
         (i i (+ i 1))
         (res res (* res i)))
      ((> i n) res)))
  (loop n 1 1))
```

Источник < https://bmstu-iu9.github.io/scheme-labs/lect03.html>

Билет 18

Экзаменационный билет №18
по курсу «Основы информатики»

1. Принцип построения лексического анализатора.
2. Основные управляющие конструкции языка Scheme.
3. Задача

Задача №18
к экзаменационному билету
по курсу «Основы информатики»

Используя БНФ, запишите формальную грамматику для логических выражений, записанных в традиционной инфиксной нотации. Грамматика должна учитывать приоритет операторов. В выражениях могут встречаться: константы `True` и `False`, операторы `and`, `or` (бинарные), `not` (унарный), круглые скобки, переменные. Имена переменных могут состоять только из латинских букв. Приоритет операторов убывает в ряду: `not`, `and`, `or`. Порядок вычислений может задаваться круглыми скобками так же, как в арифметических выражениях.
Приведите пример такого выражения и дерево разбора этого выражения.

2. Основные управляющие конструкции языка Scheme.

В программе используются три основные управляющие конструкции: следование (***begin*** в Scheme), ветвление (***if*** и ***cond*** в Scheme) и цикл (***do*** в Scheme).

Источник < https://bmstu-iu9.github.io/scheme-labs/lect01.html>

Следование Begin:

Если мы имеем вызов вида (f (g …)) то в Scheme гарантируется, что сначала вычисляется (g …), а потом (f …).

Но если мы имеем вызов вида (f (g …) (h …)) то, что выполнится раньше — g или h — зависит от реализации. Разные реализации Scheme могут вычислять аргументы справа налево или слева направо.

Но если нужно вывести на печать несколько значений, то порядок вызова будет существенен: функции должны вызваться в правильном порядке.В Scheme есть особая форма (begin …), гарантирующая порядок вычисления:

```
(begin
  (display "Hello, ")
  (display "World!"))
```

begin выполняет действия в том порядке, в котором они записаны.

Результатом begin'а является результат последнего действия.

```
(begin (* 7 3) (+ 6 4))              → 10
```

Источник < https://bmstu-iu9.github.io/scheme-labs/lect05.html>

Ветвление if cond

If:

```
(if <условие>
    <если истина, по умолчанию #t>
    <если ложь, по умолчанию #f>)
```

Если <условие> - ложь, то вычисляется <выражение-2> и становиться результатом (if…), если не ложь - <выражение-1>

Cond:

```
(cond
  (<условие 1> <выражение>)
  (<условие 2> <выражение>)
  (else <выражение>)
```

*else может отсутствовать

- По очереди вычисляются условия
- Если условие k - не #f, результатом (cond …) станет результат выражения k
- Если все условия -#f, результат (cond …) - выражение E

Источник < https://bmstu-iu9.github.io/scheme-labs/lect02.html>

Цикл do:

```
(do ((<перем> <нач> <модиф [необ.]>)
     ...
     (<перем> <нач> <модиф [необ.]>))
    (<усл. выраж.> <возврат [необ.]>)
   <выраж>
   ...
   <выраж>)
```

Выполнять действия, пока не будет выполнено условие

Источник < https://bmstu-iu9.github.io/scheme-labs/lect05.html>

3. Задача

```
<логическое выражение> ::= <логическое выражение> "OR" <логическое выражение>
  | <логическое выражение> "AND" <логическое выражение>
  | "NOT" <логическое выражение>
  | (<логическое выражение>)
  | <выражение>
<выражение> ::= <переменная> | <константа>
<переменная> ::= '<символ>{<символ>...}'
<символ> ::= A | … | Z | a | … | z
<константа> ::= "True" | "False"
```

Пример:
C and (Aa and True or (not B))
Derivation:

```
<логическое выражение> => <логическое выражение> and <логическое выражение>
 => <выражение> and <логическое выражение>
 => <переменная> and <логическое выражение>
 => <символ> and <логическое выражение>
 => <C> and <логическое выражение>
 => <C> and (<логическое выражение>)
 => <C> and (<логическое выражение> or <логическое выражение>)
 => <C> and (<логическое выражение> and <логическое выражение> or <логическое выражение>)
 => <C> and (<выражение> and <логическое выражение> or <логическое выражение>)
 => <C> and (<переменная> and <логическое выражение> or <логическое выражение>)
 => <C> and ('<символ><символ>' and <логическое выражение> or <логическое выражение>)
 => <C> and ('A<символ>' and <логическое выражение> or <логическое выражение>)
 => <C> and (Aa and <логическое выражение> or <логическое выражение>)
 => <C> and (Aa and <выражение> or <логическое выражение>)
 => <C> and (Aa and <константа> or <логическое выражение>)
 => <C> and (Aa and True or <логическое выражение>)
 => <C> and (Aa and True or (<логическое выражение>))
 => <C> and (Aa and True or (not <логическое выражение>))
 => <C> and (Aa and True or (not <выражение>))
 => <C> and (Aa and True or (not <переменная>))
 => <C> and (Aa and True or (not <символ>))
 => <C> and (Aa and True or (not B))
```

1. Принцип построения лексического анализатора

Грамматика для стадии лексического анализа описывается, как правило, без рекурсии (имеется ввиду, без не хвостовой рекурсии), т.к. лексическая структура языка не требует вложенных конструкций.

Назначение лексического анализа: 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: строка символов (или список символов), выходные: последовательность токенов. Можно сказать, что дерево разбора для грамматики лексем вырожденное — рекурсия есть только по правой ветке (cdr).

Пример построения лексического анализатора:

I. Первая фаза написание парсера — построение LL(1)-грамматики.

II. Вторая фаза — механистическое построение парсера по грамматике.

III. Третья фаза — реализация семантических действий.

Источник < https://bmstu-iu9.github.io/scheme-labs/lect13.html>

Билет 19

Экзаменационный билет №19
по курсу «Основы информатики»

1. Принцип построения нисходящего синтаксического анализатора, осуществляющего разбор методом рекурсивного спуска.
2. Ассоциативные списки.
3. Задача

Задача №19
к экзаменационному билету
по курсу «Основы информатики»

На языке Scheme напишите определение процедуры, возвращающей число своих вызовов.

1. 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.

Построенный синтаксический анализатор выдаёт сообщение о принадлежности входной строки к заданному языку.

Написание синтаксического анализатора состоит из этапов:

1. Составление LL(1)-грамматики для данного языка программирования.
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.

Синтаксический анализ:

Его грамматика как правило описывается уже с использованием рекурсии, дерево разбора рекурсивное.

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

Входные данные: список токенов, выходные: дерево разбора (или синтаксическое дерево).

Дерево разбора — дерево, построенное для данной грамматики и данной входной строки, такое что, корнем является аксиома грамматики, листьями — символы входной строки, внутренними узлами являются нетерминальные символы грамматики, потомки внутренних узлов упорядочены и соответствуют правилам грамматики, при перечислении листьев слева-направо получаем исходную строку.

Источник < https://bmstu-iu9.github.io/scheme-labs/lect13.html>

2. 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))
```

Источник < https://bmstu-iu9.github.io/scheme-labs/lect08.html>

3. Задача

```
(define counter  0)
(define (fynction)
  (set! counter  (+ counter  1))
  counter)

(fynction)
(fynction)
(fynction)
```

Билет 20

Экзаменационный билет №20
по курсу «Основы информатики»

1. Основные понятия объектно-ориентированного программирования.
2. Точечные пары и списки в языках семейства Lisp.
3. Задача

Задача №20
к экзаменационному билету
по курсу «Основы информатики»

На языке Python напишите определение функции, возвращающей число своих вызовов.

1. Основные понятия объектно-ориентированного программирования

Объектно-ориентированное программирование (ООП) — программа пишется как набор взаимодействующих друг с другом объектов. Объект объединяет в себе данные и поведение (код), объекты могут посылать друг другу сообщения.

Источник < https://bmstu-iu9.github.io/scheme-labs/lect01.html>

Объектно-ориентированное программирование - парадигма программирования, при которой программы представляют собой набор объектов, принадлежащих различным классам, взаимодействующих между собой.

Основные понятия:

I. *Класс* - некоторый шаблон, описывающий устройство объектов.
II. *Объект* - конкретный экземпляр класса.
III. *Инкапсуляция* - принцип ООП, согласно которому класс реализуется как "чёрный ящик", т.е. пользователь класса может с ним взаимодействовать только посредством интерфейса класса.
IV. *Полиморфизм* - возможность обрабатывать данные разных типов одним методом.

2. Точечные пары и списки в языках семейства Lisp

Точечная пара - основная структура данных в языках семейства Lisp. Она имеет два поля - голову и хвост.

Списки представляют собой пары, в хвосте которых находятся пары, но последняя пара имеет в хвосте нулевой элемент. Если у последней пары в хвосте ненулевой элемент, то такой список называется неправильным.

Пример: Правильный список (a b c) представляет собой (a . (b . (c . 0))) Неправильный список (a b . c) представляет собой (a . (b . c))

Основные функции:

1. **Cons:** (cons <голова> <хвост>) → <список>
2. **List:** (list <элемент> <элемент> <элемент>) → <список>
3. **Car:** (car <список>) → <голова списка>
4. **Cdr:** (cdr <список>) → <хвост списка>
5. **Null?:** (null? <список>) → <bool>

3. Задача

```
count = 0
def function():
    global count
    count += 1
    return count

print(function())
print(function())
print(function())
```

Билет 21

Экзаменационный билет №21
по курсу «Основы информатики»

1. Файловая система, путь к файлу и атрибуты файла. Основные команды оболочки для работы с файлами и папками.
2. Общая характеристика языка Python (Javascript), типизация и система типов.
3. Задача

Задача №21
к экзаменационному билету
по курсу «Основы информатики»

На языке Python напишите определение класса, реализующего абстрактный тип данных «очередь».

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>

3. Задача (Python)

```
class Queue:

    def __init__(self):
        self.queue = list()

    def add_element(self,val):
        self.queue.insert(0,val)
        return True

    def top_elem(self):
        if len(self.queue) > 0:
            return self.queue[0]

    def remove_element(self):
        if len(self.queue)>0:
            self.queue.pop()

q = Queue()
q.add_element(1)
q.add_element(2)
print(q.top_elem())
q.remove_element()
print(q.top_elem())
```

2. Общая характеристика языка Python (Javascript), типизация и система типов

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.

Типы:

- неизменяемые — int, float, bool, str, tuple, complex;
- изменяемые — list, set, dict

Билет 22

Экзаменационный билет №22
по курсу «Основы информатики»

1. Стандартные потоки ввода-вывода, аргументы командной строки. Перенаправление ввода-вывода в командной оболочке Bash, использование конвейеров (pipes).
2. Особенности присваивания, копирования и передачи в функцию объектов в языке Python (Javascript).
3. Задача

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

1) Стандартные потоки ввода-вывода

Всё в Linux — это файлы, в том числе — ввод и вывод. Операционная система идентифицирует файлы с использованием дескрипторов.

Каждому процессу позволено иметь до девяти открытых дескрипторов файлов. Оболочка bash резервирует первые три дескриптора с идентификаторами 0, 1 и 2. Вот что они означают.

- 0, STDIN — стандартный поток ввода.
- 1, STDOUT — стандартный поток вывода.
- 2, STDERR — стандартный поток ошибок.

STDIN — это стандартный поток ввода оболочки. Для терминала стандартный ввод — это клавиатура. Когда в сценариях используют символ перенаправления ввода — <, Linux заменяет дескриптор файла стандартного ввода на тот, который указан в команде. Система читает файл и обрабатывает данные так, будто они введены с клавиатуры.

STDOUT — стандартный поток вывода оболочки. По умолчанию это — экран. Большинство bash-команд выводят данные в STDOUT, что приводит к их появлению в консоли. Данные можно перенаправить в файл, присоединяя их к его содержимому, для этого служит команда >>

Источник <https://habr.com/ru/company/ruvds/blog/326594/>

2) Конвейеры (pipes)

Конвейеры - это команды, которые соединены операторами ;, &&, ||для выполнения в определенной последовательности. Операторы организации конвейеров работают следующим образом:

- команда1 ; команда2- команда2 выполняется после команды1 независимо от результата её работы ;
- команда1 && команда2- команда2 выполняется только после успешного выполнения команды1(то есть с кодом завершения 0);
- команда1 || команда2- команда2 выполняется только после неудачного выполнения команды1 (то есть код завершения команды1будет отличным от 0)

Источник <https://proglib.io/p/bash-notes2>

Задача №22
к экзаменационному билету
по курсу «Основы информатики»

На языке Python напишите определение класса, реализующего абстрактный тип данных «стек».

3. Задача (Python)

```
class Stack:

    def __init__(self):
        self.stack = list()

    def add_element(self,val):
        self.stack.append(val)
        return True

    def top_elem(self):
        if len(self.stack) > 0:
            return self.stack[len(self.stack) - 1]

    def remove_element(self):
        if len(self.stack)>0:
            self.stack.pop()

q = Stack()
q.add_element(1)
q.add_element(2)
print(q.top_elem())
q.remove_element()
print(q.top_elem())
```

2. Особенности присваивания, копирования и передачи в функцию объектов в языке Python (Javascript).

Оператор присваивания в Python не создаёт копию объекта, а создаёт новую переменную, присваивая ей ссылку на переданный объект, поэтому после изменения одного объекта, изменятся также будет и другой.

Чтобы создать копию объекта с новой ссылкой используется модуль *copy* и функции copy() и deepcopy().

copy копирует объект, но оставляет элементы в нем теми же ссылками, поэтому если будет происходить изменение внутри объекта (например, изменение элементов list), то они также будут меняться в обоих объектах.

deepcopy же создает полную копию, и уже даже элементы list будут скопированы и лежать по другим ссылкам. Новый объект не будет зависим от старого.

Передача в функции

В питоне всё есть объект, объекты бывают изменяемые и неизменяемые. Передача изменяемых объектов происходит как будто по ссылке, неизменяемых - по значению.

- неизменяемые — int, float, bool, str, tuple, complex;
- изменяемые — list, set, dict.

Подробнее:
Источник <https://python-school.ru/blog/pass-by-assignment/>

Билет 23

Экзаменационный билет №23
по курсу «Основы информатики»

1. Командный интерпретатор Bash: общая характеристика языка, пример сценария командной оболочки.
2. Действия программиста для создания сценария («скрипта») на интерпретируемом языке программирования, предназначенного для запуска из командной оболочки UNIX-подобной операционной системы.
3. Задача

Задача №23
к экзаменационному билету
по курсу «Основы информатики»

На языке Python (Javascript) напишите 2 варианта функции, вычисляющей среднее арифметическое последовательности чисел. В одном варианте используйте императивные управляющие конструкции, во втором – встроенные функции высших порядков.

1. 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."
```

2. 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. Пример для Python:

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

Далее нужно сделать текстовый файл со скриптом исполняемым командой chmod .

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

```
Теперь файл доступен для запуска.
```

3. Задача (Python)

```
def arifm_imperate(xs):
    summ = 0
    kol = 0
    for x in xs:
        summ += x
        kol += 1
    return summ / kol

def arifm_high_func(xs):
    return sum(xs) / len(xs)

print(arifm_imperate([1, 2, 5, 6]))
print(arifm_high_func([1, 2, 5, 6]))
```

Билет 24

Экзаменационный билет №24
по курсу «Основы информатики»

1. Понятие объекта. Создание и использование объектов в языке Python (Javascript).
2. Функциональное программирование и обработка последовательностей: применение функций высшего порядка для обработки последовательностей на языке Python (Javascript).
3. Задача

Задача №24
к экзаменационному билету
по курсу «Основы информатики»

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)`

1. **Понятие объекта. Создание и использование объектов в языке Python (Javascript).**

Объект в Python – это набор данных (переменных) и методов (функций), к оторые воздействуют на эти данные

Всё в питоне является объектами (переменные, списки и т.д), при этом объекты можно создавать самостоятельно. Для этого используется ключевое слово *class.*

Классы имеют 3 основные компоненты:

Атрибут:
Атрибут — это элемент класса. Например, у прямоугольника таких 2: ширина (width) и высота (height).

Метод:
- Метод класса напоминает классическую функцию, но на самом деле — это функция класса. Для использования ее необходимо вызывать через объект.
- Первый параметр метода всегда self (ключевое слово, которое ссылается на сам класс).

Конструктор:
- Конструктор — уникальный метод класса, который называется __init__.
- Первый параметр конструктора во всех случаях self (ключевое слово, которое ссылается на сам класс).
- Конструктор нужен для создания объекта.
- Конструктор передает значения аргументов свойствам создаваемого объекта.
- В одном классе всегда только один конструктор.
- Если класс определяется не конструктором, Python предположит, что он наследует конструктор родительского класса.

Пример:

```
# Прямоугольник.
class Rectangle :
    'Это класс Rectangle'
    # Способ создания объекта (конструктор)
    def __init__(self, width, height):
        self.width= width
        self.height = height

    def getWidth(self):
        return self.width

    def getHeight(self):
        return self.height

    # Метод расчета площади.
    def getArea(self):
        return self.width * self.height
```

Источник < https://pythonru.com/osnovy/klass-i-obekt-v-python>

2. **Функциональное программирование и обработка последовательностей: применение функций высшего порядка для обработки последовательностей на языке Python (Javascript)**

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

`lambda x: x ** 2 - безымянная функция, возводящая число в квадрат.`

Функция filter принимает функцию предикат и итератор, возвращает итератор, элементами которого являются данные из исходного итератора, для которых предикат возвращает True:

Пример:
```
>>> list(filter(lambda x: x > 0, [-1, 1, -2, 2, 0]))
[1, 2]
```

Функция map принимает функцию и итератор, возвращает итератор, элементами которого являются результаты применения функции к элементам входного итератора.
Пример:
```
a = [1, 2, 3, 4, 5]
>>> list(map(lambda x: x**2, a))
[1, 4, 9, 16, 25]
```

3. **Задача**

```
(define (drop xs n)
  (if (or (= n 0) (null? xs))
      xs
      (drop (cdr xs) (- n 1))))

(drop '(1 2 3 4) 2)
```

Билет 25

Экзаменационный билет №25
по курсу «Основы информатики»

1. Стандартные потоки ввода-вывода, аргументы командной строки. Перенаправление ввода-вывода в командной оболочке Bash, использование конвейеров (pipes).
2. Средства метапрограммирования языка Python (Javascript).
3. Задача

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

1) Стандартные потоки ввода-вывода

Всё в Linux — это файлы, в том числе — ввод и вывод. Операционная система идентифицирует файлы с использованием дескрипторов.

Каждому процессу позволено иметь до девяти открытых дескрипторов файлов. Оболочка bash резервирует первые три дескриптора с идентификаторами 0, 1 и 2. Вот что они означают.

- 0, STDIN — стандартный поток ввода.
- 1, STDOUT — стандартный поток вывода.
- 2, STDERR — стандартный поток ошибок.

STDIN — это стандартный поток ввода оболочки. Для терминала стандартный ввод — это клавиатура. Когда в сценариях используют символ перенаправления ввода — <, Linux заменяет дескриптор файла стандартного ввода на тот, который указан в команде. Система читает файл и обрабатывает данные так, будто они введены с клавиатуры.

STDOUT — стандартный поток вывода оболочки. По умолчанию это — экран. Большинство bash-команд выводят данные в STDOUT, что приводит к их появлению в консоли. Данные можно перенаправить в файл, присоединяя их к его содержимому, для этого служит команда >>

Источник <https://habr.com/ru/company/ruvds/blog/326594/>

2) Конвейеры (pipes)

Конвейеры - это команды, которые соединены операторами ;, &&, ||для выполнения в определенной последовательности. Операторы организации конвейеров работают следующим образом:

- команда1 ; команда2- команда2 выполняется после команды1 независимо от результата её работы ;
- команда1 && команда2- команда2 выполняется только после успешного выполнения команды1(то есть с кодом завершения 0);
- команда1 || команда2- команда2 выполняется только после неудачного выполнения команды1 (то есть код завершения команды1будет отличным от 0)

Источник <https://proglib.io/p/bash-notes2>

2. 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`.

Пример:

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

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

3. Задача

Для этого билета нет уникальной задачи.

Билет 26

Экзаменационный билет №26
по курсу «Основы информатики»

1. Символьный тип в языке Scheme и его применение.
2. Особенности присваивания, копирования и передачи в функцию объектов в языке Python (Javascript).
3. Задача

1. Символьный тип в языке Scheme и его применение.

Символьный тип данных — это «зацитированное», «замороженное» имя:
Имеет известный нам предикат типа и функции преобразования:

```
(symbol? 'hello)              → #t
(symbol? "hello")             → #f
(symbol? #\a)                 → #f
(symbol->string 'hello)       → "hello"
(string->symbol "hello")      → hello
(string->symbol
  (string-append
    (symbol->string 'hell)
    (symbol->string 'o))))    → hello
```

Примеры использования:

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

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".

Применяется в основном для превращения программы в данные, и последующего их исполнения.

Источник < https://bmstu-iu9.github.io/scheme-labs/lect08.html>

2. Особенности присваивания, копирования и передачи в функцию объектов в языке Python (Javascript).

Оператор присваивания в Python не создаёт копию объекта, а создаёт новую переменную, присваивая ей ссылку на переданный объект, поэтому после изменения одного объекта, изменятся также будет и другой.

Чтобы создать копию объекта с новой ссылкой используется модуль *copy* и функции copy() и deepcopy().

copy копирует объект, но оставляет элементы в нем теми же ссылками, поэтому если будет происходить изменение внутри объекта (например, изменение элементов list), то они также будут меняться в обоих объектах.

deepcopy же создает полную копию, и уже даже элементы list будут скопированы и лежать по другим ссылкам. Новый объект не будет зависим от старого.

Передача в функции

В питоне всё есть объект, объекты бывают изменяемые и неизменяемые. Передача изменяемых объектов происходит как будто по ссылке, неизменяемых - по значению.

- неизменяемые — int, float, bool, str, tuple, complex;
- изменяемые — list, set, dict.

Подробнее:
Источник <https://python-school.ru/blog/pass-by-assignment/>

3. Задача

Для этого билета нет уникальной задачи.