

Домашние задания

Домашнее задание 1. Обработка ошибок

1. Добавьте в программу, вычисляющую выражения, обработку ошибок, в том числе:
 - о ошибки разбора выражений;
 - о ошибки вычисления выражений.
2. Для выражения $1000000 * x * x * x * x * x / (x - 1)$ вывод программы должен иметь следующий вид:

x	f
0	0
1	division by zero
2	32000000
3	121500000
4	341333333
5	overflow
6	overflow
7	overflow
8	overflow
9	overflow
10	overflow

Результат `division by zero (overflow)` означает, что в процессе вычисления произошло деление на ноль (переполнение).

3. При выполнении задания следует обратить внимание на дизайн и обработку исключений.
4. Человеко-читаемые сообщения об ошибках должны выводиться на консоль.
5. Программа не должна «вылетать» с исключениями (как стандартными, так и добавленными).

[Репозиторий курса](#)

Домашнее задание 2. Бинарный поиск

1. Реализуйте итеративный и рекурсивный варианты бинарного поиска в массиве.
2. На вход подается целое число x и массив целых чисел a , отсортированный по невозрастанию. Требуется найти минимальное значение индекса i , при котором $a[i] \leq x$.
3. Для `main`, функций бинарного поиска и вспомогательных функций должны быть указаны, пред- и постусловия. Для реализаций методов должны быть приведены доказательства соблюдения контрактов в терминах троек Хоара.
4. Интерфейс программы.
 - о Имя основного класса — `search.BinarySearch`.
 - о Первый аргумент командной строки — число x .
 - о Последующие аргументы командной строки — элементы массива a .
5. Пример запуска: `java search.BinarySearch 3 5 4 3 2 1`. Ожидаемый результат: 2.

Домашнее задание 3. Очередь на массиве

1. Определите модель и найдите инвариант структуры данных «[очередь](#)». Определите функции, которые необходимы для реализации очереди. Найдите их пред- и постусловия, при условии что очередь не содержит `null`.
2. Реализуйте классы, представляющие **циклическую** очередь на основе массива.
 - о Класс `ArrayQueueModule` должен реализовывать один экземпляр очереди с использованием переменных класса.
 - о Класс `ArrayQueueADT` должен реализовывать очередь в виде абстрактного типа данных (с явной передачей ссылки на экземпляр очереди).

- Класс `ArrayQueue` должен реализовывать очередь в виде класса (с неявной передачей ссылки на экземпляр очереди).
 - Должны быть реализованы следующие функции (процедуры) / методы:
 - `enqueue` — добавить элемент в очередь;
 - `element` — первый элемент в очереди;
 - `dequeue` — удалить и вернуть первый элемент в очереди;
 - `size` — текущий размер очереди;
 - `isEmpty` — является ли очередь пустой;
 - `clear` — удалить все элементы из очереди.
 - Модель, инвариант, пред- и постусловия записываются в исходном коде в виде комментариев.
 - Обратите внимание на инкапсуляцию данных и кода во всех трех реализациях.
3. Напишите тесты к реализованным классам.

Домашнее задание 4. Очереди

1. Определите интерфейс очереди `Queue` и опишите его контракт.
2. Реализуйте класс `LinkedListQueue` — очередь на связном списке.
3. Выделите общие части классов `LinkedListQueue` и `ArrayQueue` в базовый класс `AbstractQueue`.

Это домашнее задание связано с предыдущим.

Домашнее задание 5. Вычисление в различных типах

Добавьте в программу разбирающую и вычисляющую выражения трех переменных поддержку вычисления в различных типах.

1. Создайте класс `expression.generic.GenericTabulator`, реализующий интерфейс `expression.generic.Tabulator`:

```
public interface Tabulator {
    Object[][][] tabulate(
        String mode, String expression,
        int x1, int x2, int y1, int y2, int z1, int z2
    ) throws Exception;
}
```

Аргументы

- `mode` — режим работы

Режим Тип

i int с детекцией переполнений
 d double
 bi [BigInteger](#)

- `expression` — вычисляемое выражение;
- `x1, x2; y1, y2; z1, z2` — диапазоны изменения переменных (включительно).

Возвращаемое значение — таблица значений функции, где `R[i][j][k]` соответствует `x = x1 + i, y = y1 + j, z = z1 + k`. Если вычисление завершилось ошибкой, в соответствующей ячейке должен быть `null`.

2. Доработайте интерфейс командной строки:

- Первым аргументом командной строки программа должна принимать указание на тип, в котором будут производиться вычисления:

Опция Тип

-i int с детекцией переполнений
 -d double

-bi [BigInteger](#)

- Вторым аргументом командной строки программа должна принимать выражение для вычисления.
 - Программа должна выводить результаты вычисления для всех целочисленных значений переменных из диапазона $-2..2$.
3. Реализация не должна содержать [непроверяемых преобразований типов](#).
 4. Реализация не должна использовать аннотацию [@SuppressWarnings](#).
 5. При выполнении задания следует обратить внимание на простоту добавления новых типов и операций.

Домашнее задание 6. Функциональные выражения на JavaScript

1. Разработайте функции `const`, `variable`, `add`, `subtract`, `multiply`, `divide`, `negate` для вычисления выражений с тремя переменными: `x`, `y` и `z`.
2. Функции должны позволять производить вычисления вида:

```
let expr = subtract(
    multiply(
        const(2),
        variable("x")
    ),
    const(3)
);

println(expr(5, 0, 0));
```

При вычислении выражения вместо каждой переменной подставляется значение, переданное в качестве соответствующего параметра функции `expr`. Таким образом, результатом вычисления приведенного примера должно быть число 7.

3. Тестовая программа должна вычислять выражение $x^2 - 2x + 1$, для `x` от 0 до 10.
4. **Сложный вариант.** Требуется дополнительно написать функцию `parse`, осуществляющую разбор выражений, записанных в [обратной польской записи](#). Например, результатом

```
parse("x x 2 - * x * 1 +")(5, 0, 0)
```

должно быть число 76.

5. При выполнении задания следует обратить внимание на:
 - Применение функций высшего порядка.
 - Выделение общего кода для операций.

Домашнее задание 7. Объектные выражения на JavaScript

1. Разработайте классы `Const`, `Variable`, `Add`, `Subtract`, `Multiply`, `Divide`, `Negate` для представления выражений с тремя переменными: `x`, `y` и `z`.

1. Пример описания выражения $2x - 3$:

```
let expr = new Subtract(
    new Multiply(
        new Const(2),
        new Variable("x")
    ),
    new Const(3)
);

println(expr.evaluate(5, 0, 0));
```

2. При вычислении такого выражения вместо каждой переменной подставляется её значение, переданное в качестве аргумента метода `evaluate`. Таким образом, результатом вычисления приведенного примера должно стать число 7.

3. Метод `toString()` должен выдавать запись выражения в обратной польской записи.

Например, `expr.toString()` должен выдавать «2 x * 3 -».

2. Функция `parse` должна выдавать разобранное объектное выражение.

3. **Сложный вариант.**

Метод `diff("x")` должен возвращать выражение, представляющее производную исходного выражения по переменной `x`. Например, `expr.diff("x")` должен возвращать выражение, эквивалентное `new Const(2)` (выражения `new Subtract(new Const(2), new Const(0))` и

```
new Subtract(
  new Add(
    new Multiply(new Const(0), new Variable("x")),
    new Multiply(new Const(2), new Const(1))
  )
  new Const(0)
)
```

так же будут считаться правильным ответом).

4. **Бонусный вариант.** Требуется написать метод `simplify()`, производящий вычисления константных выражений. Например,

```
parse("x x 2 - * 1 *").diff("x").simplify().toString()
```

должно возвращать «x x 2 - +».

5. При выполнении задания следует обратить внимание на:

- Применение инкапсуляции.
- Выделение общего кода для операций.
- Минимизацию необходимой памяти.

Домашнее задание 8. Обработка ошибок на JavaScript

1. Добавьте в предыдущее домашнее задание функцию `parsePrefix(string)`, разбирающую выражения, задаваемые записью вида «(- (* 2 x) 3)». Если разбираемое выражение некорректно, метод `parsePrefix` должен бросать человеко-читаемое сообщение об ошибке.
2. Добавьте в предыдущее домашнее задание метод `prefix()`, выдающий выражение в формате, ожидаемом функцией `parsePrefix`.
3. При выполнении задания следует обратить внимание на:
 - Применение инкапсуляции.
 - Выделение общего кода для операций.
 - Минимизацию необходимой памяти.
 - Обработку ошибок.

Домашнее задание 9. Линейная алгебра на Clojure

1. Разработайте функции для работы с объектами линейной алгебры, которые представляются следующим образом:
 - скаляры – числа
 - векторы – векторы чисел;
 - матрицы – векторы векторов чисел.
2. Функции над векторами:
 - `v+/v-/v*/v/d` – покомпонентное сложение/вычитание/умножение/деление;
 - `scalar/vect` – скалярное/векторное произведение;
 - `v*s` – умножение на скаляр.
3. Функции над матрицами:
 - `m+/m-/m*/m/d` – поэлементное сложение/вычитание/умножение/деление;
 - `m*s` – умножение на скаляр;
 - `m*v` – умножение на вектор;
 - `m*m` – матричное умножение;
 - `transpose` – транспонирование;
4. **Сложный вариант.**

1. Ко всем функциям должны быть указаны контракты. Например, нельзя складывать вектора разной длины.
2. Все функции должны поддерживать произвольное число аргументов. Например `(v+ [1 2] [3 4] [5 6])` должно быть равно `[9 12]`.
5. При выполнении задания следует обратить внимание на:
 - Применение функций высшего порядка.
 - Выделение общего кода для операций.
6. **Code Golf**

Правила

1. Выигрывает самая короткая программа. Длина программы считается после удаления незначимых пробелов.
2. Можно использовать произвольные функции [стандартной библиотеки Clojure](#).
3. Нельзя использовать функции Java и внешних библиотек.
4. Подача решений через [чат](#). Решение должно быть корректно отформатировано и начинаться с `;Solution` номинация длина. Например, `;Solution det-2 1000`.

Номинации

- `det-3` — определитель матрицы за $O(n^3)$;
- `det-s` — определитель дольше чем за $O(n^3)$;
- `inv-3` — обратная матрица за $O(n^3)$;
- `inv-s` — обратная дольше чем за $O(n^3)$.

Домашнее задание 10. Функциональные выражения на Clojure

1. Разработайте функции `constant`, `variable`, `add`, `subtract`, `multiply`, `divide` и `negate` для представления арифметических выражений.
 1. Пример описания выражения $2x-3$:


```
(def expr
  (subtract
    (multiply
      (constant 2)
      (variable "x"))
    (constant 3)))
```
 2. Выражение должно быть функцией, возвращающей значение выражения при подстановке переменных, заданных отображением. Например, `(expr {"x" 2})` должно быть равно 1.
2. Разработайте разборщик выражений, читающий выражения в стандартной для Clojure форме. Например,


```
(parseFunction "(- (* 2 x) 3)")
```

 должно быть эквивалентно `expr`.
3. **Сложный вариант.** Функции `add`, `subtract`, `multiply` и `divide` должны принимать произвольное число аргументов. Разборщик так же должен допускать произвольное число аргументов для `+`, `-`, `*`, `/`.
4. При выполнении задания следует обратить внимание на:
 - Выделение общего кода для операций.

Домашнее задание 11. Объектные выражения на Clojure

1. Разработайте конструкторы `Constant`, `Variable`, `Add`, `Subtract`, `Multiply`, `Divide` и `Negate` для представления арифметических выражений.
 1. Пример описания выражения $2x-3$:

```
(def expr
  (Subtract
    (Multiply
      (Constant 2)
      (Variable "x"))
    (Constant 3)))
```

2. Функция (`evaluate expression vars`) должна производить вычисление выражения `expression` для значений переменных, заданных отображением `vars`. Например, (`evaluate expr {"x" 2}`) должно быть равно 1.
3. Функция (`toString expression`) должна выдавать запись выражения в стандартной для Clojure форме.
4. Функция (`parseObject "expression"`) должна разбирать выражения, записанные в стандартной для Clojure форме. Например,

```
(parseObject "(- (* 2 x) 3)")
```

должно быть эквивалентно `expr`.

5. Функция (`diff expression "variable"`) должна возвращать выражение, представляющее производную исходного выражения по заданной переменной. Например, (`diff expression "x"`) должен возвращать выражение, эквивалентное (`Constant 2`), при этом выражения (`Subtract (Constant 2) (Constant 0)`) и

```
(Subtract
  (Add
    (Multiply (Constant 0) (Variable "x"))
    (Multiply (Constant 2) (Constant 1)))
  (Constant 0))
```

так же будут считаться правильным ответом.

2. **Сложный вариант.** Конструкторы `Add`, `Subtract`, `Multiply` и `Divide` должны принимать произвольное число аргументов. Разборщик так же должен допускать произвольное число аргументов для `+`, `-`, `*`, `/`.
3. При выполнении задания можно использовать функции, приведённые на лекции.
4. При выполнении задания можно использовать любой способ представления объектов.

Домашнее задание 12. Комбинаторные парсеры

1. **Простой вариант.** Реализуйте функцию (`parseObjectSuffix "expression"`), разбирающую выражения, записанные в суффиксной форме, и функцию `toStringSuffix`, возвращающую строковое представление выражения в этой форме. Например,

```
(toStringSuffix (parseObjectSuffix "( ( 2 x * ) 3 - )"))
```

должно возвращать `((2 x *) 3 -)`.

2. **Сложный вариант.** Реализуйте функцию (`parseObjectInfix "expression"`), разбирающую выражения, записанные в инфиксной форме, и функцию `toStringInfix`, возвращающую строковое представление выражения в этой форме. Например,

```
(toStringInfix (parseObjectInfix "2 * x - 3"))
```

должно возвращать `((2 * x) - 3)`.

3. **Бонусный вариант.** Добавьте в библиотеку комбинаторов возможность обработки ошибок и продемонстрируйте ее использование в вашем парсере.
4. Функции разбора должны базироваться на библиотеке комбинаторов, разработанной на лекции.

Домашнее задание 13. Простые числа на Prolog

1. Разработайте правила:

- `prime(N)`, проверяющее, что N – простое число.
- `composite(N)`, проверяющее, что N – составное число.
- `prime_divisors(N, Divisors)`, проверяющее, что список `Divisors` содержит все простые делители числа N , упорядоченные по возрастанию. Если N делится на простое число P несколько раз, то `Divisors` должен содержать соответствующее число копий P .

2. Варианты

- Простой: $N \leq 1000$.
- Сложный: $N \leq 10^5$.
- Бонусный: $N \leq 10^7$.

3. Вы можете рассчитывать, на то, что до первого запроса будет выполнено правило `init(MAX_N)`.

Домашнее задание 14. Деревья поиска на Prolog

1. Реализуйте ассоциативный массив (`map`) на основе деревьев поиска. Для решения можно реализовать любое дерево поиска логарифмической высоты.

2. Простой вариант. Разработайте правила:

- `map_build(ListMap, TreeMap)`, строящее дерево из упорядоченного списка пар ключ-значение ($O(n)$);
- `map_get(TreeMap, Key, Value)`, проверяющее, что массив содержит заданную пару ключ-значение ($O(\log n)$).

3. Сложный вариант. Дополнительно разработайте правила:

- `map_put(TreeMap, Key, Value, Result)`; добавляющее пару ключ-значение в массив, или заменяющее текущее значение для ключа ($O(\log n)$);
- `map_remove(TreeMap, Key, Result)` удаляющее отображение для ключа ($O(\log n)$);
- `map_build(ListMap, TreeMap)`, строящее дерево из **не**упорядоченного списка пар ключ-значение ($O(n \log n)$).