**Домашнее задание 1. Обработка ошибок**

1. Добавьте в программу вычисляющую выражения обработку ошибок, в том числе:
   * ошибки разбора выражений;
   * ошибки вычисления выражений.
2. Для выражения 1000000\*x\*x\*x\*x\*x/(x-1) вывод программы должен иметь следующий вид:
3. x f
4. 0 0
5. 1 division by zero
6. 2 32000000
7. 3 121500000
8. 4 341333333
9. 5 overflow
10. 6 overflow
11. 7 overflow
12. 8 overflow
13. 9 overflow
14. 10 overflow

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

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

[Тесты к домашним заданиям](https://www.kgeorgiy.info/git/geo/paradigms-2020)

**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] <= x.
3. Для 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.
   * Имя основного класса — BinarySearch.
   * Первый аргумент командной строки — число x.
   * Последующие аргументы командной строки — элементы массива a.
5. Пример запуска: java BinarySearch 3 5 4 3 2 1. Ожидаемый результат: 2.

**Домашнее задание 3. Очередь на массиве**

1. Найдите инвариант структуры данных «[очередь](http://ru.wikipedia.org/wiki/%D0%9E%D1%87%D0%B5%D1%80%D0%B5%D0%B4%D1%8C_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5))». 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 с применением массива.
   * Класс ArrayQueueModule должен реализовывать один экземпляр очереди с использованием переменных класса.
   * 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.
   * 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 LinkedQueue — очередь на связном списке.
3. Выделите общие части классов LinkedQueue и ArrayQueue в базовый класс AbstractQueue.

**Домашнее задание 5. Вычисление в различных типах**

1. Добавьте в программу разбирающую и вычисляющую выражения поддержку различных типов.
   * Первым аргументом командной строки программа должна принимать указание на тип, в котором будут производится вычисления:

|  |  |
| --- | --- |
| **Опция** | **Тип** |
| -i | int |
| -d | double |
| -bi | [BigInteger](http://docs.oracle.com/javase/8/docs/api/java/math/BigInteger.html) |

* + Вторым аргументом командной строки программа должна принимать выражение для вычисления.
  + Реализация не должна содержать [непроверяемых преобразований типов](http://docs.oracle.com/javase/specs/jls/se8/html/jls-5.html#jls-5.1.9).
  + Реализация не должна использовать аннотацию [@SuppressWarnings](http://docs.oracle.com/javase/specs/jls/se8/html/jls-9.html#jls-9.6.4.5).

1. При выполнении задания следует обратить внимание на легкость добавления новых типов и операциий.

**Домашнее задание 6. Функциональные выражения на JavaScript**

1. Разработайте функции cnst, variable, add, subtract, multiply, divide, negate для вычисления выражений с одной переменной.
2. Функции должны позволять производить вычисления вида:
3. let expr = subtract(
4. multiply(
5. cnst(2),
6. variable("x")
7. ),
8. cnst(3)
9. );
10. println(expr(5));

При вычислении такого выражения вместо каждой переменной подставляется значение, переданное в качестве параметра функции expr (на данном этапе имена переменных игнорируются). Таким образом, результатом вычисления приведенного примера должно стать число 7.

1. Тестовая программа должна вычислять выражение x2−2x+1, для x от 0 до 10.
2. **Сложный вариант.** Требуется написать функцию parse, осуществляющую разбор выражений, записанных в [обратной польской записи](http://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%80%D0%B0%D1%82%D0%BD%D0%B0%D1%8F_%D0%BF%D0%BE%D0%BB%D1%8C%D1%81%D0%BA%D0%B0%D1%8F_%D0%B7%D0%B0%D0%BF%D0%B8%D1%81%D1%8C). Например, результатом

parse("x x 2 - \* x \* 1 +")(5)

должно быть число 76.

1. При выполнение задания следует обратить внимание на:
   * Применение функций высшего порядка.
   * Выделение общего кода для бинарных операций.

**Домашнее задание 7. Объектные выражения на JavaScript**

1. Разработайте классы Const, Variable, Add, Subtract, Multiply, Divide, Negate для представления выражений с одной переменной.
   1. Пример описания выражения 2x-3:
   2. let expr = new Subtract(
   3. new Multiply(
   4. new Const(2),
   5. new Variable("x")
   6. ),
   7. new Const(3)
   8. );

* 1. Метод evaluate(x) должен производить вычисления вида: При вычислении такого выражения вместо каждой переменной подставляется значение x, переданное в качестве параметра функции evaluate (на данном этапе имена переменных игнорируются). Таким образом, результатом вычисления приведенного примера должно стать число 7.
  2. Метод toString() должен выдавать запись выражения в [обратной польской записи](http://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%80%D0%B0%D1%82%D0%BD%D0%B0%D1%8F_%D0%BF%D0%BE%D0%BB%D1%8C%D1%81%D0%BA%D0%B0%D1%8F_%D0%B7%D0%B0%D0%BF%D0%B8%D1%81%D1%8C). Например, expr.toString() должен выдавать 2 x \* 3 -.

1. **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).

Функция parse должна выдавать разобранное объектное выражение.

1. **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 - +».

1. При выполнение задания следует обратить внимание на:
   1. Применение инкапсуляции.
   2. Выделение общего кода для операций.

**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 выполнение задания следует обратить внимание на:
   * Применение инкапсуляции.
   * Выделение общего кода для бинарных операций.
   * Обработку ошибок.
   * Минимизацию необходимой памяти.

**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\* – покоординатное сложение/вычитание/умножение;
   * scalar/vect – скалярное/векторное произведение;
   * v\*s – умножение на скаляр.
3. Функции над матрицами:
   * m+/m-/m\* – поэлементное сложение/вычитание/умножение;
   * m\*s – умножение на скаляр;
   * m\*v – умножение на вектор;
   * m\*m – матричное умножение;
   * transpose – траспонирование;
4. **Сложный вариант.**
   * Ко всем функциям должны быть указаны контракты. Например, нельзя складывать вектора разной длины.
   * Все функции должны поддерживать произвольное число аргументов. Например (v+ [1 2] [3 4] [5 6]) должно быть равно [9 12].
5. При выполнение задания следует обратить внимание на:
   * Применение функций высшего порядка.
   * Выделение общего кода для операций.

**Домашнее задание 10. Функциональные выражения на Clojure**

1. Разработайте функции constant, variable, add, subtract, multiply и divide для представления арифметических выражений.
   1. Пример описания выражения 2x-3:
   2. (def expr
   3. (subtract
   4. (multiply
   5. (constant 2)
   6. (variable "x"))
   7. (constant 3)))

* 1. Выражение должно быть функцией, возвращающей значение выражение при подстановке элементов, заданных отображением. Например, (expr {"x" 2}) должно быть равно 1.

1. Разработайте разборщик выражений, читающий выражения в стандартной для Clojure форме. Например,

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

должно быть эквивалентно expr.

1. **Сложный вариант.** Функции add, subtract, multiply и divide должны принимать произвольное число аргументов. Разборщик так же должен допускать произвольное число аргументов для +, -, \*.
2. При выполнение задания следует обратить внимание на:
   1. Выделение общего кода для операций.

**Домашнее задание 11. Объектные выражения на Clojure**

1. Разработайте конструкторы Constant, Variable, Add, Subtract, Multiply и Divide для представления выражений с одной переменной.
   1. Пример описания выражения 2x-3:
   2. (def expr
   3. (Subtract
   4. (Multiply
   5. (Constant 2)
   6. (Variable "x"))
   7. (Const 3)))

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

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

должно быть эквивалентно expr.

* 1. Функция (diff expression "variable") должена возвращать выражение, представляющее производную исходного выражения по заданой пермененной. Например, (diff expression "x") должен возвращать выражение, эквивалентное (Constant 2), при этом выражения (Subtract (Const 2) (Const 0)) и
  2. (Subtract
  3. (Add
  4. (Multiply (Const 0) (Variable "x"))
  5. (Multiply (Const 2) (Const 1)))
  6. (Const 0))

так же будут считаться правильным ответом.

1. **Сложный вариант.** Констуркторы Add, Subtract, Multiply и Divide должны принимать произвольное число аргументов. Разборщик так же должен допускать произвольное число аргументов для +, -, \*, /.
2. При выполнение задания можно использовать любой способ преставления 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 -).

1. **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).

1. **Бонусный вариант.** Добавьте в библиотеку комбинаторов возможность обработки ошибок и продемонстрируйте ее использование в вашем парсере.
2. 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 ≤ 1000.
   * Сложный: N ≤ 105.
   * Бонусный: N ≤ 107.
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*)).