Домашнее задание 1. Хэширование

1. Разработайте Java-программу `CalcMD5`, которая подсчитывает MD5-хэши файлов.
2. Программа должна принимать один аргумент командной строки — имя файла, в котором содержатся имена файлов, для которых требуется подсчитать хэши. Файлы перечислены по одному на строке.
3. Программа должна выдать на стандартный вывод MD5-хэши файлов в порядке их перечисления во входном файле. Хэши должны выдаваться в виде 32-значных шестнадцатеричных чисел.
4. Например, если файл `input.txt` содержит только `input.txt` (9 символов), то при запуске `java CalcMD5 input.txt`, на консоль должно быть выведено `A8546347050ADC932FBEC189DC9FD50D`.
5. *Примечания.*
    1. Стандартная библиотека Java содержит реализацию алгоритма MD5.
    2. Вы можете рассчитывать, что все файлы помещаются в память.
    3. Можно написать решение, состоящее из четырех содержательных строк.

Тесты к домашним заданиям

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». Определите функции, которые необходимы для реализации очереди. Найдите их пред- и постусловия.
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 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 выражений

1. Разработайте классы `Const`, `Variable`, `Add`, `Subtract`, `Multiply`, `Divide` для вычисления выражений с одной переменной.
2. Классы должны позволять составлять выражения вида

```
new Subtract(
    new Multiply(
        new Const(2),
        new Variable("x")
    ),
    new Const(3)
).evaluate(5)
```

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

3. Для тестирования программы должен быть создан класс `Main`, который вычисляет значение выражения x^2-2x+1, для `x`, заданного в командной строке.
4. При выполнение задания следует обратить внимание на:
   - Выделение общего интерфейса создаваемых классов.
   - Выделение абстрактного базового класса для бинарных операций.

Домашнее задание 6. Разбор выражений

1. Доработайте предыдущее домашнее задание, так что бы выражение строилось по записи вида

```
x * (x - 2)*x + 1
```

2. В записи выражения могут встречаться: умножение `*`, деление `/`, сложение `+`, вычитание `-`, унарный минус `-`, целочисленные константы (в десятичной системе счисления, которые помещаются в 32-битный знаковый целочисленный тип), круглые скобки, переменные (`x`) и произвольное число пробельных символов в любом месте (но не внутри констант).
3. Приоритет операторов, начиная с наивысшего
   1. унарный минус;
   2. умножение и деление;
   3. сложение и вычитание.
4. Разбор выражений рекомендуется производить методом рекурсивного спуска. Алгоритм должен работать за линейное время.

Домашнее задание 7. 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:

```
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 выводится на консоль.
5. Программа не должна «вылетать» с исключениями (как стандартными, так и добавленными).

Домашнее задание 8. Вычисление в различных типах

1. Добавьте в программу 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`

   - Реализация не должна содержать непроверяемых преобразований типов.

Домашнее задание 1. Хэширование

Домашнее задание 2. Бинарный поиск

Домашнее задание 3. Очередь на

- Реализация не должна использовать аннотацию `@SuppressWarnings`.

2. При выполнении задания следует обратить внимание на легкость добавления новых типов и операций.

Домашнее задание 9. 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:

```
var expr = subtract(
    multiply(
        cnst(2),
        variable("x")
    ),
    cnst(3)
);
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.
3. Тестовая программа должна вычислять выражение x^2-2x+1, для `x` от 0 до 10.
4. **Усложненный вариант.** Требуется написать функцию `parse`, осуществляющую разбор выражений, записанных в обратной польской записи. Например, результатом

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

   должно быть число `76`.
5. При выполнение задания следует обратить внимание на:
   - Применение функций высшего порядка.
   - Выделение общего кода для бинарных операций.

Домашнее задание 10. 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`:

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

   2. 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.
   3. Метод `toString()` должен выдавать запись выражения в обратной польской записи. Например, `expr.toString()` должен выдавать `2 x * 3 -`.
2. **Усложненный 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.

3. **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 - +`».

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

Домашнее задание 11. 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 12. 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. **Усложненный 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 выполнение задания следует обратить внимание на:
   - Применение функций высшего порядка.
   - Выделение общего кода для операций.

Домашнее задание 13. Функциональные выражения на Clojure

1. Разработайте функции `constant`, `variable`, `add`, `subtract`, `multiply` и `divide` 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 выражение при подстановке элементов, 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. **Усложненный вариант.** Функции `add`, `subtract`, `multiply` и должны принимать произвольное число аргументов. Разборщик так же должен допускать произвольное число аргументов для `+`, `-`, `*`.
4. При выполнение задания следует обратить внимание на:

- Выделение общего кода для операций.

Домашнее задание 14. 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`:

```
(def expr
  (Subtract
    (Multiply
      (Constant 2)
      (Variable "x"))
    (Const 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")` должена возвращать выражение, представляющее производную исходного выражения по заданой перменной. Например, `(diff expression "x")` должен возвращать выражение, эквивалентное `(Constant 2)`, при этом выражения `(Subtract (Const 2) (Const 0))` и

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

    так же будут считаться правильным ответом.
2. **Усложненный вариант.** Функция `(parseObjectInfix "expression")` должна разбирать выражения, записанные в инфиксной форме. Например,

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

    должно быть эквивалентно `expr`.
3. При выполнение задания можно использовать любой способ преставления объектов.