Assignment 1: Calculator

This assignment is programmed in **Rust**.

Prerequisites

- To build this *calculator* the Rust Toolchain is required.

Alternative: Using Docker

How to build the executable

```
cargo build --release
```

The compiled executable is saved in `./target/release/`.

How to Run

Just execute the `main` executable.

```
./target/release/main
# Show help
./target/release/main --help
```

Using Docker

```
# Building Docker Image
docker build -t g5a1:latest .
# Executing Docker Image
docker run --rm -it g5a1:latest
# Show help
docker run --rm -it g5a1:latest /opt/bin/calculator --help
```

Architecture

The Calculator is basically a library encapsulating all the internals of the calculator. This library is used by the main.rs executable, which initiates the default program and the other examples.

Inside the library we have the basic `Value` enumeration type. This type is used as the basic *memory cell* elements of the calculator. It can hold two different types of values - the `Single(f64)` holding a 64-bit floating point value and `List(String)` representing a *List* element consisting of a single String.

The generic `Calculator` structure type holds the different parts like the *Data Stack*, *Command Stream*, *Operation mode* variable, etc. We use a *Generic* type here to be able to *Unit test* the code without requiring `STDIN` and `STDOUT`/`STDERR` during those tests. For convenience we defined here two type alias `StdCalculator<O>` as the "Release Version" of the calculator and the

`TestCalculator` type for internal *Unit Tests*, which is only compiled in "Debug mode", but skipped in "Release mode".

Next the *Operation modes*: We defined the *Trait* (similar to an Interface type) `OperationMode<R,W>`, which defines a common interface between all operation modes. This makes it possible to make use of polymorphism in the Calculator.

Test protocol

The testing of the calculator is completely automated. For executing them see below. Those unit tests are separated into two groups: 1. Internal unit tests, for validating the internal behaviour of the calculator. 2. System tests, for validating example programs.

To execute all tests run

```
cargo test
```

We are documenting only the System tests here.

Test case identification	Description	Test result
`test::sum_triangle_surface_0`	Testing of program in register D with 0 triangles.	OK
`test::sum_triangle_surface_1`	Testing of program in register D with 1 triangles.	OK
`test::sum_triangle_surface_3`	Testing of program in register D with 3 triangles.	OK
`test::octahedron`	Testing of program in register C with edge length 5.	OK
`test::surface_triangle`	Testing of program in register B with a simple triangle.	OK