# CS100 Homework 10 (Spring, 2022)

Deadline: 2022-06-21 23:59:59

**Late submission will open for 24 hours after the deadline, with -50% point deduction.**

## Polynomial Computation

You have two tasks in this problem. First you need to implement a polynomial class which supports some basic operations. Then you need to parse an expression that contains multiple polynomial and evaluate it at an arbitray point $x \in \mathbb{R}$.

### Polynomial Class

We use a list of coefficients to represent a polynomial. For example, the polynomial $1 + 2x + 3x^4$ is represented by `[1,2,0,0,3]`, and stored as a `private` variable `m_coeffs` of type `std::vector<double>`.

You need to implement all the member functions in the `Polynomial` class. After implementing `Matrix` and `Array` in the previous homework, you should be quite familiar with those constructors and operators by now. Don't forget to use `std::move` for rvalue reference in constructors.

The class also supports reading coefficients from a file and get initialized. The `data/` folder contains the data that we need. Each file `p*.txt` contains a line of spaced polynomial coefficients, with the first one being the coefficient of the zero degree term. For example the line `-1 0 2` indicates the polynomial $-1 + 2x^2$.

Please read the comments in `include/polynomial.hpp` for more details. You need to implement all the member functions in `src/polynomial.cpp`.

### Reverse Polish Notation

In this task, we will use Reverse Polish Notation (also known as postfix notation) to represent expressions in which the operator is placed after the arguments begin operated on. For example, the expression "(3 + 4) $*$ 5" is written "3 4 + 5 $*$" in reverse polish notation. In comparision, reverse Polish notation lead to faster calculation. The reason is that reverse Polish calculators do not need expressions to be parenthesized, so fewer operations need to be entered.

In practice RPN can be conveniently evaluated using a stack structure. Reading the expression from left to right, the following operations are performed:

1. If a value appears next in the expression, **push** this value onto the stack.

2. If an operator appears next, **pop** two items from the top of the stack and push the result of the operation onto the stack.

The procedure of evaluating a RPN expression $3\ 4\ +\ 5\ *$ is as follows:

1. Push 3 onto the stack.
2. Push 4 onto the stack.
3. Pop 4 and 3 from the stack, compute $3 + 4$ and push the result onto the stack.
4. Push 5 onto the stack.
5. Pop 5 and 7 from the stack, compute $7 * 5$ and push the result onto the stack.
6. The final result is 35

In our case, an example of polynomial expression is $p_1\ p_2\ +\ p_3\ *$ where $p_1, p_2, p_3$ stands for different polynomials. Each operator and operand is separated by a space. It is guaranteed that the expression is valid and only contains supported operations for `polynomial` (addition, subtraction and multiplication). You may use `std::stack` for stack operations.

## Evaluation

We want to evaluate a function of polynomials at an arbitrary point. For example, we want to evaluate $p_1\ p_2\ +\ p_3\ *$ at point $x = 1.5$. One possible way could be

1. Read the coefficients of polynomials from file.
2. Parse the function string $p_1\ p_2\ +\ p_3\ *$ and generate a new `Polynomial`.
3. Evaluate the new polynomial at given point.

The main drawback of the approach above is that it seems difficult, if not impossible, to do polynomial division for the `Polynomial` class or on the polynomial ring.

As a remedy, we promote the use of **lambda function**. For that purpose you have to write the lambda functions for addition, multiplication etc. Then you can parse the function string, compute and store the generated function for further evaluation.

In this task, you need to implement the two functions `compute_polynomial` and `compute_lambda`. You need to implement the first solution in `compute_polynomial` which returns a `Polynomial` instance. You need to use `lambda` function in `compute_lambda` for polynomial operations and return a function of type `std::function<double(double)>`. Please read the comments in `include/polynomial_parser.hpp` for details. You need to put your implementation in `src/polynomial_parser.cpp`.

## Testing

We have also provided you some test code. Under folder `test/` you will see three files for test. The test can be performed by CTest in CMake. The expected output is written in `test/CMakeLists.txt`. You can run the test by running `cmake test` after you have successfully built your code.

You can add more tests by yourself by modifying `test/CMakeLists.txt` and adding more data in `data/`. Make sure you have correctly set the expected output or you may waste lots of time for debugging.

## Submission Guideline

We will use Autolab to judge your code. You need to sumbit a .tar file which directly contains `polynomial.cpp` and `polynomial_parser.cpp`. DO NOT add any other files or folders. The structure of your submission should be like

hw10.tar

- polynomial.cpp
- polynomial_parser.cpp