# 1 Simple Function Evaluator

This program consists of a simple actor with a command-line interface to interact with it.

- Implement an actor for the function $a_0 \cdot x^4 + a_1 \cdot x^3 + a_2 \cdot x^2 + a_3 \cdot x + a_4$. The values $a_0 ... a_4$ are bound when spawning a new actor. It then waits for messages of type `{'calc', X}`, where `'calc'` is the atom ”calc” and `X` is a value of type double. The actor replies with `{X, Y}`, where `X` is the received $x$ value and `Y` the result of the function evaluation.
- Implement a `main` that reads values for $a_0 ... a_4$ from `std::cin` and spawns an actor for the function evaluation. Afterwards, the `main` function reads a value for $x$ from `std::cin`, sends it to the actor and then awaits and prints the result, before reading the next value for $x$.

# 2 Simple Chat

The Simple Chat is text-based command-line chat consisting of a server and a client application.

## 2.1 Server

- Implement an actor that receives *join*, *chat*, *leave*, and *ls* messages. On a *join* messages (that also contains a nickname and a handle to the new actor), the sender of the message will be added to the list of chat participants. On a *chat* message, the message is forwarded to all participants except for the sender of the chat message. On a *leave* message (that may contain a goodbye message), the sender of the *leave* message is erased from the list of chat participants. The actor should also monitor all chat participants and treat *down* messages in the same way *leave* messages are handled. On an *ls* message, the actor responds with a list of known nicknames to the sender.
- Implement a `main` that publishes the implemented actor on the port specified in the command-line arguments.

## 2.2 Client

The client is a simple command-line interface to the chat. On startup, the program asks the user for a nickname that is used to register at the chat server. The chat program has the following interface:

- `"/ls"`: Lists all connected peers with their nickname.
- `"/quit"`: Terminates the program.
- Any message that does not begin with `'/'` is a chat message.
- All received messages are printed to the terminal.

# 3 Distributed Function Evaluation

Add the following two commands to the Simple Chat program:

- `"/calc X on NICKNAME for A0 A1 A2 A3 A4"`: Sends a message the client registered as NICKNAME. The receiving client then spawns a function evaluator using the values for $a_0 ... a_4$ and sends it a calculate message for X. The result, however, should be received by the sender of the original message and printed to the screen before accepting new input.

  mmm

# 4 Fortune Server

- Implement a broker that listens to a predefined port. On each new connection, it immediately sends a random “fortune” as HTTP response. A fortune is a sayings like those found on fortune cookies. The broker should have a predefined set of fortunes it can choose from. Test your broker using a web browser. Reloading the page should show a different fortune. *Hint: you can implement this broker based on the* `simple_http_broker` *example.*

- Add a new message type to your broker to add new fortunes at runtime. Publish your broker at a different port—modeling a separate control channel—using `io::publish` und add new fortunes from remote nodes.

# 5 Rendezvous Processes

- Implement a service for function evaluation (again based on the Simple Function Evaluator). Workers are being announced at a *group* by a manager via an `'idle'` message. This announcement is repeated periodically, e.g., once per second. Any listener of the group receiving the idle messages can send a request to the manager.
- The manager dispatches an incoming job to its worker and sends a `'busy'` message to the group to inform clients that is worker is no longer available. Any additional job is refused until the worker finished its job and is being announced again. Clients must never speak directly to the worker nor do workers send the result to the client. Instead, all communication is routed through the manager.
- Spawn 10 pairs of workers and managers as well as 100 clients with one job request each (only clients currently looking for available workers are subscribed to the group). The program should terminate after each client has successfully dispatched its job and has received its result.
- *Hint:* you can spawn managers and workers using the *hidden* flag to have them ignored by `await_all_actors_done()`.