Exercise sheet 4 2022-11-17

Due date: 2022-11-24 16:59

The goal of this exercise sheet is to work with standard library features, exceptions, functors and lambdas.

Exercise 1:

Complete the implementation of a cute little stack-based virtual machine.

The architecture is as follows:

- The program code is read-only, and consists of **instructions**
- The **program counter** (`pc`) determines which instruction of the program code to execute next
- The machine has a **stack** where all computations are performed
  - Instructions read and consume the topmost stack contents
  - Instructions put new items on the stack as results
  - `TOS` denotes the Top Of Stack
  - `TOS1` is the second top-most stack item

Code executed by `vm::run` is given as a string with one instruction per line, for example:

```
LOAD_CONST 622
LOAD_CONST 432
ADD
PRINT
```

First, `622` and then `432` is put on the stack. `ADD` consumes both and puts `1337` on the stack. `PRINT` takes the `1337` from the stack and prints it to `stdout`!

The provided `struct vm_state` in file `vm.h` stores VM execution state:

```
1  struct vm_state {
2      size_t pc = 0;             // execution position in program code
3      std::stack<item_t> stack;  // execution data stack
4
5      // registered instructions:
6      std::unordered_map<std::string, op_id_t> instruction_ids;
7      std::unordered_map<op_id_t, op_action_t> instruction_actions;
8  };
```

We provide you with a parser (`vm::assemble`) that converts the input code to a `std::vector` of instructions, which is the program code then.

What each instruction does, and how the instructions are executed one after the other is missing, though.

Start by looking at `vm.h` to understand how the VM works and what all the functions shall do.

You have to complete the VM implementation:

- Complete `vm::run` function so it can execute instructions and then return the result:
  - return the `TOS` value and
  - return the result string created by `WRITE` instructions (see below for a description of `WRITE`)
- Complete `vm::register_instruction` so instruction implementations can be added to a VM
- The VM must throw an exception (`throw vm_segfault{};`) if a non-existent program address is executed
- Implement and register the instructions specified below in `vm::create_vm`
  → If not enough stack items are available for the current instruction,
  `throw vm_stackfail{"optional message"};`

The VM has to execute the following instructions correctly. Think about how you can simplify repeating actions like getting items from the stack!

- `LOAD_CONST <number>` push a number to the stack
- `PRINT` print `TOS` to `stdout` [we provide this as example code]
- `EXIT` stop VM execution, return `TOS` as execution result
- `POP` remove the `TOS` item from the stack
- `ADD` calculate `TOS1 + TOS`, consume them, and put the result on the stack
- `DIV` same, just calculate `TOS1 / TOS`
  ⇒ if `TOS` is zero, throw an exception (`div_by_zero`)!
- `EQ` consume `TOS` and `TOS1` and push `1` if they were equal, otherwise `0`
- `NEQ` same, but push `0` if they were equal, `1` if equal
- `DUP` copy `TOS` by pushing it on the stack
- `JMP <addr>` set `pc` given address to jump there
- `JMPZ <addr>` if `TOS` is `0`, consume it and set `pc` to `addr`
- `WRITE` append `TOS` as number to the VM output string
- `WRITE_CHAR` append `TOS` as ASCII to the VM output string

You can of course add more cool instructions as you like :)

You're given the API we test against in `vm.h`, and our implementation skeleton in `vm.cpp`. There's some helper code in `util.h` and `util.cpp`.