Programming Languages and Paradigms
Spring 2022

Assignment 2

Submission Details

You can submit your solutions to this assignment in the OLAT course of PLP until Wednesday, Apr 13, 2022, at 11:59pm. Submit a single .zip file that includes all relevant files.

Note the following:

1. Source code files you submit should compile/run without errors.
2. Also include compiled binaries if there are any.
3. For each task, also submit screenshots of the task compiling and then running in your command line, IDE, or programming environment. For small tasks, feel free to include a single screenshot showing multiple tasks running one after another.
4. It's up to you how exactly you implement the programs internally, as long as you're using the assigned programming language. The tasks only describe the expected behavior of the programs.

1. Task: Snippets

Implement the following problems. You may implement all of them in one scope/file, embed them in a class or similar structure or have them in one scope or several files – it's up to you. Feel free to implement helper functions as needed.

Potential solutions to most of these tasks can probably be found online. You are encouraged to reach your own solution first, but you are allowed to copy from the internet at no penalty. If your solution is heavily inspired by an online solution, please provide a comment with the source url and a short explanation how the code works.

a). Quicksort

Implement the Quicksort algorithm. Note the following guidelines:

- Your solution need not be efficient. Try finding the most elegant and/or idiomatic solution. Try to understand why it may not be the most efficient and leave an appropriate comment in the source code.
- If your language supports something like generic types and/or a comparison interface that multiple types could implement, then make sure your implementation works with different types. If your language does not, implement Quicksort at least for integers, floating point numbers and strings (using the rules outlined at `https://en.wikipedia.org/wiki/Alphabetical_order#Ordering_in_the_Latin_script`) for sorting strings if the language has no built-in way of ordering strings.
- Call your implementation with a few examples and print the results.

b). Call an external command

Call an external command and retrieve it's output. Call one of these depending on your operating system, they should be available globally (if not, specify the full path to the executable in your source):

- Mac or Linux: `uname -a` (typically at `/usr/bin/uname`)
- Windows: `systeminfo` (typically at `C:\Windows\System32\systeminfo.exe`)

Print up to 10 lines of the output received from the command.

c). External Libraries

Write and execute/call a procedure which will download the following JSON[1] file and parse it.

`http://www.uvek-gis.admin.ch/BFE/ogd/52/Solarenergiepotenziale_Gemeinden_Daecher_und_Fassaden.json`

The file reports on the solar energy potential of swiss municipalities. The program should calculate the total potential of all municipalities based on the `Scenario3_*` field. Furthermore, it should print the municipality name and canton of the municipality with the *3rd*-largest potential based on this scenario.

If possible, use existing libraries (built-in or external) to perform the download and to parse the JSON.

[1] `https://www.json.org/json-en.html`

2. Task: State Machine

In this task, you will implement a finite state machine and a parser to build arbitrary state machine configurations.

A finite state machine consists of a number of states and of transitions between these states. See https://en.wikipedia.org/wiki/Finite-state_machine for a brief introduction.

Have a look at the `vending.machine` file. It contains *state* and *transition* definitions. These definitions, which don't need to appear in any particular order, define the layout of the state machine using the syntax outlined below.

States definitions. A state definition primarily consists of a name and an arbitrary text. There is also an indicator whether the state is a start or end state. See this example:

```
@*Ready{ You are standing in front of the vending machine
 [Pay] Put some money into the machine
 [Exit] Leave the machine }
```

A state definition starts on a new line with an `@` symbol as the first character. After that, it contains the following parts:

1. For *exactly one* state in the machine file, a ⋆ symbol indicates that this is the starting state. Alternatively, for *one or more* states, a + symbol indicates that this is an end state.
2. The state name, which shall only contain alphanumerical symbols and no spaces
3. An arbitrary character sequence (which can span several lines) enclosed in `{` and `}`. You can assume that `{` and `}` will not appear inside the state text.

When a state is entered, the state text (including newlines) should be printed on screen. The user can then input an action to take. If the action is valid, i.e. if a matching transition exists, the transition is executed and the next state is entered. If the action is not valid, a brief error message should be printed and the current state should be re-entered.

Transition definitions. A transition definition consists of the source and destination state names, an action name that triggers the transition, as well as an arbitrary text. See this example:

```
> Ready (Pay) Select: You put some money into the machine
```

Transition definitions occupy a single line. They start with a > symbol as the first character, followed by:

1. the source state name
2. an action name which could consist of multiple words, enclosed in `(` braces `)`
3. the destination state name
4. a colon
5. an arbitrary sequence of characters that is printed when the transition happens

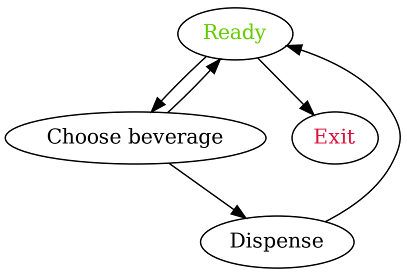
You can assume that `>`, `:`, `(` and `)` are only used as delimiters and never in the names of states or actions. They could, however, appear in the transition text following the colon. Note that whitespace around the state names or the action string before the colon should be ignored.

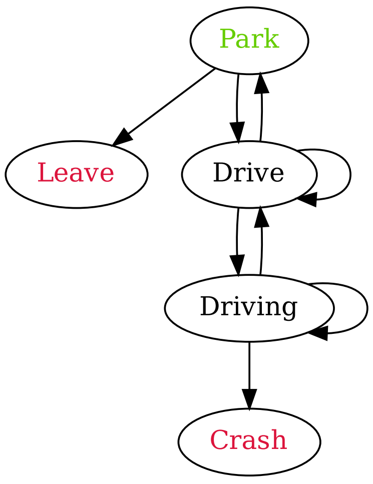
When a transition is executed, its text should be printed.

Task. Implement a program that parses a state machine configuration and executes it interactively. The program should take one positional parameter specifying the path to a file containing the state machine configuration.

Implementation Notes.

- Have a look at the other `*.machine` files. Some of them illustrate invalid examples.
- The parser should ignore anything outside valid state or transition definitions. There is no special comment indicator. If a character is not part of a state or transition, it can be simply discarded.
- The parser should reject machine files that are invalid.
- It can happen that the user chooses an invalid action. You can imagine invalid actions as arrows going from the state to itself (you always end up in the same state). This way, input error handling is just another kind of (implicit) transition. But you don't have to implement it this way.
- It might be helpful if you also implement a corresponding serializer (the opposite of the parser) so you can print your internal representation of states and transitions (basically generating a matching `.machine` file) for easier debugging. But this is not mandatory.
- Think of how to best model the states and transitions. Maybe you can use a class or an algebraic data type, depending on your programming language. In a functional language, any new state should probably be the output of a function taking an old state and an input.
- You may draw inspiration from implementations you find online. Please specify the sources in your code when doing so.
- Below are graphical representations of `vending.machine` and `car.machine`.
- Observe the example execution on the next page.
- Note that for the third assignment, you will extend this state machine with additional features, so it's worth coming up with a not-too-hacky solution.





```
# ./state_machine vending.machine
You are standing in front of the vending machine
 [Pay] Put some money into the machine
 [Exit] Leave the machine
Smash
Invalid input!
 You are standing in front of the vending machine
 [Pay] Put some money into the machine
 [Exit] Leave the machine
Pay
You put some money into the machine
 The machine is ready to accept your choice
 [Cancel] Hit the reset button
 [Choose beverage] Select a beverage
Cancel
You cancel the transaction
 You are standing in front of the vending machine
 [Pay] Put some money into the machine
 [Exit] Leave the machine
Pay
You put some money into the machine
 The machine is ready to accept your choice
 [Cancel] Hit the reset button
 [Choose beverage] Select a beverage
Choose beverage
You select a beverage
 Your choice has been dropped into the chute
 [Take] Take the beverage from the chute
Take
You remove the beverage from the chute
 You are standing in front of the vending machine
 [Pay] Put some money into the machine
 [Exit] Leave the machine
Exit
You’re not thirsty right now
```