University of Zurich[UZH]

s.e.a.l. software evolution & architecture lab

Programming Languages and Paradigms Spring 2021

Assignment 3

Submission Details

You can submit your solutions to this assignment in the OLAT course of PLP until Wednesday, May 05, 2021, 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. 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.
3. 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: Text-based adventure game

In the previous assignment, you implemented a state machine that can parse and run `.machine` files. In this assignment, you will extend your previous solution to support an additional runtime and source file: a game engine that can run arbitrary text-based adventure games specified in a `.gfl` file format as specified below. The semantics and logic are largely the same as in the previous state machine assignment. The specification for `.glf` game files also resembles the one for `.machine` files, although they follow a different global structure.

Your solution should be based on your previous state machine implementation and extend it, so that it will now support both the new and old file/state machine formats. Change or extend the code where necessary to implement the new parser and runtime. Make an effort to *minimize code duplication* among the two parsers and runtimes.

Instead of having two spearate sections containing one-line definitions for states and transitions, `.gfl` files use the two characters `@` and `>` to indicate lines where these definitions begin. State and transition definitions may appear in any order.

States. State definitions can stretch multiple lines and consist of the following parts:

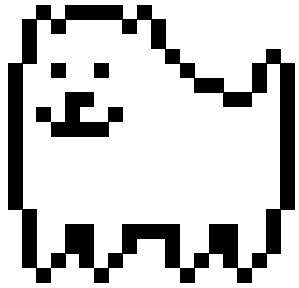
- An `@` sign at the beginning of a line is a prefix marking the beginning of a state definition. Exactly one state definition in the game file must start with `@*` to indicate the initial game state and one or more states must use the `@+` prefix to indicate end states of the game.
- The prefix is followed by the state ID followed by a colon (`:`) and newline. State IDs must be unique within a game file and can contain any except the following characters: `@:*`
- After this first line, there can be zero or more lines which are either *empty* or start with *at least one space*. These lines represent the *game screen* shown in this state. Their content should simply be printed on screen when the state is entered. The *game screen* can contain any characters (as long as the lines are empty or start with a space).

Here are two examples of valid game state definitions:

```
@Watchtower Crossroads:
                                                |>>>
                                                |
                                             _  _|_  _
                                            |;|_|;|_|;|
                                            \\.    .  /
                                             \\:  .  /
                                               ||:   |
                                               ||:.  |        \,/
                                               ||:  .|              /'\
                              _ /              ||: , |
                             <=|==>            ||:   |
                            <==|===>           ||: . |
                 __          <=|=>           _||_   |
      ____--'~    '--~~__     |       __ -----~    ~'---,              ___
 -~~~~                  ~----|_ ,--~'                  ~~-----_____-~'   '~------~~

  / Go north -> You're guessing the tower is about 20 miles away     \
  | Go west  -> This will take you to the ocean                      |
  | Go east  -> You can't see anything but desert in that direction  |
  \ Go south -> This path leads back home                           /
```

```
@*Neutral:          # Starting state as indicated by the *
```



```
  [Pet]: He's a good boy.
  [Order talk]: "Who's a good boy?"
  [Order *]: What other tricks does Doggo know?
  [Abandon]: You wouldn't! How could you?!
```

Note that the *game screen* – anything following the state ID until the next start of a state or transition definition – should simply be printed to the command line. How the *game screen* is formatted and what it includes is up to the writer. Not all available transitions from that state need to be included in the *game screen* shown as some could be hidden from the player.

Transitions. Transitions are also very similar to the previous specification. However, instead of being in their own section, transitions are prefixed with a > sign. So the syntax for a transition can be described as:

```
>State1 (Command param1 param2) State Two: What happens during the transition.
```

If invalid commands or parameters are entered, the behavior should be the same as previously (*i.e.*, print an error and return to the same state), however there exists one extension over the old syntax, namely that a command can be followed by a * sign instead of additional parameters, *i.e.,*:

```
>Select (Choose *) Select: Invalid beverage selected.
```

Such a transition shall be used as a fall back if all other parameters for this command are invalid. Until the first colon, you can assume that the characters `():*` are only used for the transition syntax, but not within state, command or parameter names. The description after the colon can contain any characters.

The following examples of valid transition definitions correspond to the choices printed to the player in the "Watchtower Crossroads" state. There are exactly 4 valid commands and if something else is entered, the generic error message should be shown:

```
>Watchtower Crossroads (Go north) Watchtower: You walk for 5 hours to reach it.
>Watchtower Crossroads (Go west) Coast: A short hike later you can smell the ocean.
>Watchtower Crossroads (Go east) DesertEvening: You've been walking until sundown.
>Watchtower Crossroads (Go south) Home: You walk back to your home town.
```

In the following example, the `Order` command takes various parameters, but if none of the ones specified match, a custom error message is printed instead of the default.

```
> Neutral (Pet) Petting: Doggo is excited!
> Neutral (Order sit) Neutral: Doggo sits down and looks at you in anticipation!
> Neutral (Order talk) Neutral: Doggo tells a joke. Nobody laughs. Doggo is happy!
> Neutral (Order turn) Neutral: Doggo spins around, trying to catch its tail!
> Neutral (Order *) Neutral: Doggo tilts its head in confusion... # custom error message
```

```
> Neutral (Order sleep) Sleeping: Doggo plays dead...
> Neutral (Abandon) GameOver: Doggo is sad to see you go :(
> Sleeping (Wake up) Neutral: Doggo wasn't actually sleeping!
```

Differences between `.machine` and `.gfl` files. In essence, the only things different in `.gfl` syntax compared to the `.machine` syntax from the previous assignment are:

- States start with `@` or `@*`, and transitions start with `>`. To parse `.gfl` files, the following strategy should work:
  1. Find any octothorpes and delete them and anything following them.
  2. Chop up the remaining text whereever `@` or `>` are at the beginning of a line.
  3. Process these segments individually depending on whether they are empty (only whitespace and newlines), a state or a transition. Strip whitespace from transition descriptions as needed.
- What used to be a one line "state description" is now a multi-line *game screen*. A *game screen* ends just before the next state or transition definition.
- Transitions support the `A (Command *) B:` syntax for custom error messages.

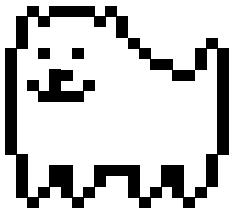
State machine The state machine itself remains unchanged. It prints the description or *game screen* and waits for input. It reads the input and picks the correct next state. Very little change to the existing implementation should be required to support the new game content.

Implementation Notes.

- The parser should ignore anything after an octothorpe (`#`), trailing whitespace and blank lines outside of game screen definitions.
- Have a look at the `doggo.gfl` and `hotel.gfl` files for full working examples. Please let me know if you think they contain bugs.
- Make reasonable assumptions (for example about whitespace) and refer to assignment 2, task 3, if you feel this specification is incomplete. Otherwise, write me an email or write on the forum for clarifications.
- You may want to clear the command line before printing the next transition description and *game screen* to make things look nicer.
- Your program should support and process files in UTF-8 if at all possible.
- See the example execution on the next page.

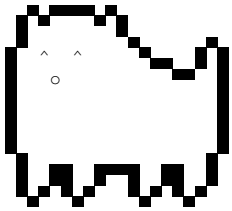
```
# ./state_machine doggo.gfl

[Neutral]
```



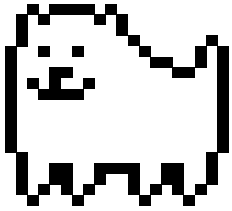
```
  [Pet]: He’s a good boy.
  [Order talk]: "Who’s a good boy?"
  [Order *]: What other tricks might Doggo know?
  [Abandon]: You wouldn’t! How could you?!

$ Pet
> Doggo is excited!
[Petting]
```




```
  [Pet]: He’s still a good boy.
  [Stop petting]: Even if it’s never enough.

$ Stop petting
> Doggo forgives you.
[Neutral]
```



```
  [Pet]: He’s a good boy.
  [Order talk]: "Who’s a good boy?"
  [Order *]: What other tricks might Doggo know?
  [Abandon]: You wouldn’t! How could you?!

$ Abandon
> Doggo is sad to see you go :(
[GameOver]
```