CS-111 Final project: A text adventure game

Overview

For this assignment, you will work in groups to make a text adventure game by adding code to the basic boiler-plate we provided in the assignment. For an example of a trivial game, see the sample-game.rkt file.

Adventure games have a long history. Classically, they consist of exploring spaces and solving puzzles. Generally, moving through the space requires solving some kind of simple puzzle. For example, some of the rooms will be accessible to you but one will be locked, so you need to look through the other rooms to find the key. Old school games often involved more moving through space and interacting with objects than interacting with characters, since the latter are much harder to program. You should probably avoid trying to implement any character AI in this class. If you do want to implement characters in your game, they should have very limited behavior.

If you aren't familiar with text adventures and want a quick primer on them and their genre conventions, take a look at Emily Short's *Welcome to Interactive Fiction*. If you're a fan and want to read some scholarly work on adventure games and interactive fiction more generally, take a look at *The IF Theory Reader*.

If, after finishing this project, you find yourself wanting to write more games like this but don't want to have to deal with Racket code, take a look at Inform 7, which is an interactive fiction language that reads like English. If you don't care about implementing objects that do things and just want to write some short stories you can click through, take a look at Twine. If you love Visual Novels and are interested in learning some python, take a look at RenPy.

Grading

This assignment will be in two parts.

First you will write and submit your game, and then you will peer review the games of three other groups. Your final grade for the first part will be the median of the grades given to you by the reviewers. The grading rubric will basically be to check that you wrote all the code you were supposed to write (see below) and that it seems to work. If it does, then you will get full credit.

You will also be graded for the peer reviews, but they will count much less. Review grading will be based simply on whether you completed the reviews.

YOU WILL NOT BE GRADED ON WHETHER YOUR GAME IS FUN, ARTISTIC, OR OTHERWISE FULFILLING. YOU DO NOT NEED TO STRESS.

Groups

You can work in groups of up to 4. You may not work in groups of 5. You can also do the project on your own, but we don't recommend it as it is likely to be somewhat more work than if you worked in a group. That said, the expectations for the game will scale with the number of people in the group.

We have created a **#final-project-group-formation** channel on Discord should you want to search for group members.

You will receive instructions later on for how to register your group members with canvas.

Getting help

Feel free to post questions to discord or piazza about how to implement different kinds of features for your game. We will be happy to tell you how in English. There is a special **#adventure-game-howto-questions** channel for just this purpose. If it's impossible or too ambitious, we'll tell you that too. You will then need to do the actual implementation in Racket. And you are explicitly allowed to read it to get ideas for your game. You just can't copy other people's code.

Requirements

You need to make the following:

New code

You have to write new **types**, **fields**, **procedures**, and **methods** for your game so that there can be a richer variety of objects in your game would with richer behavior than those in the code we distributed. You can add fields and methods to new types you create or you can add them to the existing types. As long as it's new code, it's fine. If you make a major change to one of the procedures or methods that we wrote, you can count that too.

Here are the minimum requirements:

Group size	New types	New fields	New procedures and/or methods	Total
1	4	4	4	15
2	7	6	7	25
3	7	6	10	35
4	8	6	12	45

So a group of 3 needs to implement 35 total types, fields, procedures, and/or methods in total. The exact numbers are up to the group, but at least 7 of those need to be new types, at least 6 new fields, and at least 10 new methods or procedures.

This might seem intimidating, but it's not as bad as it sounds. For example, you might implement eating (please don't just copy the code from the tutorial). Since most kinds of things are not edible, you make a separate **food** type that's a subtype of **prop**, and you add a **calories** field to it to track how filling it is. You might also add a field to the player to track how much they've eaten so they won't eat more after they're full. That's 4 points toward the required total. Then you need two methods for the **eat** command, one for the thing class, that just prints a message saying the object is inedible, and one for the food class, that checks if

you're full, and if not, destroys the object and increments your fullness. Plus, when you make the food type, you'll need to make a **new-food** procedure (like new-prop, new-person, etc.) to make an initialize instances of it. So that would be a total of 7 points for implementing eating.

The one thing you shouldn't do is to make many different types or methods that all basically do the same thing. For example, if you have a `flower` type, you might want to have many different flowers in your game. That's great. And it's fine if the only difference between them is what string they print when you use the `examine` command on them. But in that case, you should only have one `flower` type that has a field for what text to print, rather than making separate types for every flower, and separate examine methods for every type, just so that you can count 2 points for each flower.

You don't need to be paranoid about this. But when you write your self-assessment (see below), you'll list the different types, fields, methods, and procedures you wrote and what they do. Just don't put your peer reviewers in the position of thinking "gee, all these methods seem to basically be the same." In any case, don't stress about this – if you have questions about what's reasonable, please feel free to reach out to us.

Fill in (start-game)

You also need to make an actual world that contains rooms and things and instances of your types. Find the code for (start-game) and fill it in with the code for making your rooms and their contents.

Walkthrough

You should add at least one walkthrough, named **win**, using the define-walkthrough command:

```
(define-walkthrough win
   (go (the door))
   (eat (the banana))
   …)
```

This defines a procedure, **win**, that restarts the game and runs all the commands in order. Each time your type `(win)`, it will run a complete game for you.

You can use walkthroughs as a kind of check-expect to automatically test your game. But in any case, you should include at least the **win** walkthrough that your reviewers can consult if they get confused.

Note: although we ask you to include a walkthrough called win, the game need not have a win condition per se, if you don't want it to. You can just make an interesting environment for people to wander through.

Deliverables

You will receive instructions later on for how to register your group on canvas. You must register your group before you turn in your assignment. Once you've registered your group, you should upload to canvas a zip file containing:

- A README file that provides any necessary instructions on how to play the game.
- All the rkt files provided here
- Your game (i.e. a modified version of adventure.rkt). This can be in adventure.rkt or in a separate file, so long as it is obvious to the peer reviewers which file they're supposed to open and run.
- A filled-out copy of the Self Assessment file describing what went right with your project, what went wrong, what you learned, and what score you believe you deserve for the assignment.

Only one group member should upload the assignment.

Advice on using the adventure.rkt code

Start by reading through the code to get a sense of how it all works. This is an important skill. When you read through it, you aren't trying to understand all of it. You're trying to get a sense of what its parts are (the types, procedures, etc.) and roughly what they do. You don't need to understand all of the procedures or every line of code within a procedure. There are also a bunch of chunks of code at the end of the assignment that are magic that Ian wrote that are outside the scope of this class. Don't feel you need to understand the code in the section at the end called "Utilities". But note that there are a bunch of useful procedures in there you might want to use:

- (here)
  Tells you what room the player is in
- (stuff-here)
  Gives you a list of the things in the room, including the player
- (stuff-here-except-me)
  Gives you a list of the things in the room, not including the player.
- (my-inventory)
  The things in the player's "pockets"
- (accessible-objects)
  The objects in the current room or in the player's pockets. That's what is searched by the "the" command: when you say (the door) it scans this list for a door.
- (have? *thing*)
  Tells you if thing is in the player's inventory.
- (have-a? *predicate*)
  Tells you if the player has something that satisfies *predicate*.
- (everything)
  Finds all the objects that are potentially reachable by the player. In particular, it's the player, their location and contents (i.e. their inventory), plus all those objects' locations and contents, and all *those* object's locations and contents, *etc*. Note that when Racket prints lists containing game objects, those game objects will print strangely. So if you want to read the list, you probably want to use (print-everything) instead.
- (print-everything)
  Prints the everything list. If you make an object and it doesn't appear here, then that means you forgot to link the room its in to the player's starting room.
- (every *predicate*)
  All the objects from (everything) that satisfy predicate. It's just (filter *predicate* (everything)). Again, printing lists with game objects in them is weird, so you may prefer to use print-every.
- (print-every *predicate*)
  Prints them all.
- (display-line *any*)
  Prints its argument and then starts a new line of output.
- (string->words *string*)
  Breaks up a string into a list of its component words

- (words->string list-of-strings)
  Joins all the strings in the list, putting spaces between them.

Testing

You can test your code just by playing the game. But we **strongly** recommend you write walkthroughs to test you game. This will let you test and retest your code as you change it without having to manually type long sequences of commands.

We have also provided a **check** command that takes a Boolean and throws an exception if it's false. You can put this into your walkthroughs to have them automatically check what's happening. For example, the walkthrough for the example game tests the **take** command by first taking the banana and then including the line:

```
(check (have? (the banana))
```

After it. Adding this to the walkthrough means the walkthrough will proceed if the banana is in the player's inventory after picking it up (and hence the take command worked), but throw an exception and stop the walkthrough if for some reason the take command didn't work and the player does not have the banana.