CS403 Final Project

For the final project, you're tasked with building a command interpreter for an autonomous rover on some planet/moon in our solar system, an exoplanet, the ocean, etc. (anywhere you want).

Today, automated rovers are built with a set of commands defined and we send commands with data that get parsed into an action that the rover performs whether it be moving, drilling, or going to sleep for a set period of time.

In this project you will build something similar but in a much more simplified manner. Your task is to implement some form of cross-process communication strategy that would allow us to communicate with a rover running in one process from a controller in another process. You'll find some starter code for this in the skeleton project. It provides the ability to run multiple rovers, and send commands to each of them. Try it out before making any modifications to see how it looks. You can start the rovers by running:

python rover.py

And then send them a command with:

python main.py parsing-tests/test.txt Rover1

Or

python main.py parsing-tests/test.txt Rover2

The first parameter is the file containing the code to run, and the second argument is the name of the rover to target. Ensure that you read all the files contained in the skeleton as there are hints scattered throughout. The parsing_components.py file contains the "enhanced" nodes for doing semantic analysis, and running your code. There are some examples there of how you can proceed for this project.

Note in passing that while I have provided a skeleton parser that you can modify and use, you're allowed to build and use your own. As long as the main.py, and rover.py interfaces remain similar it will be fine. I need to be able to run the rover in one terminal, and then send commands to it through another terminal.

Be sure to provide detailed instructions for how to build, and run your project if it's not obvious. 50% will be deducted if these aren't provided and it isn't obvious how to run. Do not use packages outside of the standard available Python3 libraries.

Program (75%)

Before starting, carefully read everything below, and look at the grading criteria to know what you will be marked on. You'll have a chance to justify your choices in a report. Your goal is to allow simple commands like the following to be parsed, and run by the rover(s) to make them navigate a map with successive commands sent from the main program:

```
{
      int n = 0 ;
      while ( n < 2 ) {
          if (rover . canTurnRight ( ) ) {
              rover . turnRight ( ) ;
          } else {
              rover . warnNoRightTurn ( ) ;
          }
          n = n + 1 ;
      }
}
```

Your rover program needs to have a rover object that you'll be able to run methods on dynamically. It also needs to have some for of memory. To call these methods on the object and provide arguments you are free to pick any form of style. The example above uses the standard convention for calling methods, but using something like this would be just as valid (and may be simpler for parsing):

```
{
      rover changeMap map1;
}
```

To access attributes of the rover, use a syntax such as this or something similar to the alternative shown above:

```
{
      rover . attr . position ;
}
```

Be careful to not make an ambiguous language and ensure that the style selection you pick would always be unambiguous when the rover is accessed. To be able to run/execute this language, you will need to do some scope checking (only at the global level), and type checking. All of this should be done in three separate steps, the first of which (parsing) is already implemented: parsing -> semantics -> execution/running. You will implement the semantics stage, and then the execution stage. For the execution stage, here's a small example of how your if statement might be executed:

```
      if conditional.run(): # Returns a true or false (or something similar)
            return if_statement.run()
      elif else_statement is not None: # Else branch might not exist
            return else_statement.run()
```

I've provided **one map (map.txt)** that you can traverse with the rover and you must implement the following for its movements:

1. The rover is initialized onto a random location on the map (must not be on an X).
2. The rover must have an orientation (an integer denoting the direction, e.g. N: 0, E: 1, S:2, W:3).'
3. The rover cannot move onto a tile with an X.
4. The rover can move to any tile that is not marked with an X.
5. The rover needs to be able to drill (or perform some action), and tiles marked with D will give a reward.

Sample map:

```
                    XXXXXXXXXXXXXXXXXX
                    XXXD XX D   XXXXXX
                    XDXX X  XX  XXXXXX
                    X    X  XX  DXXXXX
                    X               X
                    XXXXXXDXXXXXXXXXX
                    XXXXXXXXXXXXXXXXXX
```

You are free to add additional tile markings. The rover needs to have the following capabilities or commands available:

1. **Info, and Queries:** The rover needs to be able to report back to the command center! Give it a simple `print` method that outputs the information into standard output (stdout, the default stream for Python's print). The rover also needs methods for querying it's position, and orientation. It should also provide a method for providing what it sees in the tile(s) directly in front of it (you can pick what the rover shows).
2. **Map Change:** Add a method to move to a random location on a new map.
3. **Movement:** Move forward, backwards, left, or right by X tiles from the current tile. Orientation needs to change as it moves.
4. **Turning:** Make the rover turn left or right on the same tile with no movement.
5. **Drilling/Main-Feature:** Command that performs a drilling on the current tile. It doesn't need to be drilling; it could be things like exploring, and building too. There are no restrictions here. Your rover simply needs to perform some operation on some marked tiles (such as the D tile).
6. **5 Features In Total:** Including the Drilling/Main-Feature above, you need to have 5 Features in total that your rover can perform on the marked tiles (10 features if you are working in a group).

All of that being said, you need to start with implementing the semantic analysis (type, and scope checking), and the code execution methods for your language. After that, you can add your robot grammar to the parse tree so that you can run its methods. (This is the suggested approach, but you might like to go a different route).

Assumptions that you can make, and tips/tricks for the project:

1. No storage between commands is necessary other than the rover's direction, its location on the map, and some attributes to track the 5 features.

2. This language has no functions, classes, or strings.

3. Input is separated by blanks and new lines. Statements end in semi-colons, or after a statement/statement-block ({<CODE>}) for statements like `if `, `while`, and `for`.

4. You can work from the grammar from Assignment #4. It's missing a small bit (the Robot AST), but it's a great starting point given the time frame. This is contained within the parser files in the skeleton project.

5. The parser.py file contains some modifications in preparation for using it for semantic analysis, and code execution. Look at the parser_components.py file for more information on this. Note that they are distinct steps.

6. Assume that there will never be variables that conflict with the names you use as the keyword methods/features of the robot and that the `rover` keyword will always uniquely refer to the robot.

7. You have the ability to run multiple rovers using the skeleton but you only need to run 1 rover for this project. You can receive a bonus for building a "rover army" (see next pages).

8. Robot and rover are used interchangeably in this document, it's your choice how you implement those keywords.

Restrictions:

1. **No special modules. Only modules from the standard python library are allowed.**

Group work:

No more than 3 people in a group. If you are 3 people in a group, then you need to implement all three points below. If you are in a group of 2, then you only need to implement two of the points below (your choice of which). Each of these can be done separately from each other to a certain extent after the base of the project is implemented:

1. **Scope Checking:** You will need to make 2 levels of scope available. This means that we should be able to re-declare a variable within a single nested block (no deeper).

2. **Type Checking:** Allow sub/super-typing in the language. This means that an int can be placed in a double variable, but a double can't be stored within an int (this is the only sub/super-typing available in the language).

3. **Robot AST and Features:** You must implement 10 features instead of 5. Up to 3 features out of 10 do not need interaction with the map.

I suggest making use of a private Github repo to coordinate changes. **For each of these that is unimplemented in the submission, up to 5% will be removed from the total grade (up to 15% in total).**

Project Requirements Summary

All of the following needs to be implemented in your project:

1. Type Checking
    - No sub/super-typing is required. Only 1:1 matches so an int shouldn't match with a double, and vice-versa.
        - **If you are working in a group, you need to allow this super/sub-typing for int, and double.**
    - Ignore the char type. Only bool, int, and double should be checked. However, the char can be checked as a Python string.
    - **Type consistency for all expressions, and assignment statements.**

2. Scope Checking
    - Only 1 level of depth required so all declarations are global. **2 levels if you are working in a group.**
    - No need to check array indexes for overflow, assume these are always correct, but do make sure that the type is an int.
    - **All the variables are declared before use.**

3. Code execution
    - Run the code directly in Python (see the MinusNode for an example)

4. Add the `rover` keyword object to the existing language to access the rover object and it's method.

5. Add the ability for the rover to navigate a map.
    - I've provided a sample map in the skeleton (map.txt)
    - **X** represents areas where you cannot navigate to.
    - **D** represents areas of interest and that you can navigate to.
    - **Blanks** represent areas that you can navigate to.

6. Add the following methods to your rover (perhaps at the same time as 5):
    - **info**: Prints the current status of the rover
    - **getters/queries**: Getters or query methods to check specific information.
    - **turnRight**: Turns the rover to the right.
    - **turnLeft**: Turns the rover to the left.
    - **move**: Moves the rover forward by one in some direction.

7. Add 5 extra features methods to your rover to access the `D` map areas and perform some operation, e.g. drilling, building, digging, exploring/discovering.
    - You are free to add extra markings in the maps as well.
    - 1 feature does not need to interact with the map.
    - **10 features if you are working in a group. Up to 3 features do not need to interact with the map.**

8. Add the ability to change the map.

9. BONUS: Run 5 rovers at once with other conditions (see next page).

Testing (10%)

You will want to check expected vs. actual values produced. Another thing to check is logging, but ensure that you make a comment about this if you are looking at the logs. That said, in a multi-process environment we are leading ourselves into non-deterministic runtime environments so it will be impossible to check expected vs actual values for all of your metrics (some will still work though).

There is only one way around this: **build unit tests for all of your code**. These tests only run a single portion of your code (e.g. one test for a single method in a class). Given the time constraint, building unit tests is unfeasible as it can take as much, or more time to write than the source code.

In this case, **you can test your program by running a handful of known successful tests, along with a collection of tests that probe edge cases in your program** (e.g. can it handle a blank command?).

Collect these tests in the **parsing-tests** folder.

Report (5%)

As a future software engineer, you need to get used to writing out your reasons for doing one thing over another. This will help you with keeping track of decision history, explaining your solution to others, and finding possible simplifications in your program. If it helps, assume that you are attempting to convince one of your managers that the way you built this program, is the only way to build it.

You will need to produce a report of at least 1 page (or 1000 words, whichever comes first) describing your project, and the idea or direction you took for the rover. This will include reasoning for things like "why I implemented the Robot AST in this way given my chosen grammar".

I expect to see all of the following in your document:

- Overview of your program.
    - Describe the idea you implemented.
    - Overview of how it's implemented including all the features you added.
- Design/implementation decisions
    - How was the Robot AST implemented
- Grammar suitable for recursive-descent parsing for your command interpreter.
    - Only include the portions needed for the Robot AST everything else should stay like

Comments, Formatting, and Readability (7.5%)

Ensure that your code is commented properly (I suggest following the PEP-8 style guidelines), along with proper formatting. You can make use of a multitude of linters to do so automatically, I suggest using Python Black.

Bonus: Rover Army (5%)

If you're feeling ambitious, the skeleton provides the ability to run multiple rovers at once. For this project, you are only required to have one running but for a 5% bonus, you can create multiple rovers and have them interact with each other in some way. To obtain the full bonus, the rovers MUST all communicate with each other, and there must be at least 5 rovers running at once. All the rovers must be able to accept commands as well. It is possible to receive partial bonus marks.

Submission Guidelines (2.5%)

Keep your code organized within a single folder so you don’t need to be concerned with import issues.

- Submit a zipped folder of the project by email with the following naming convention (using my name as example): 'MIER_GRE_FINAL.zip'
- Submission exporting guidelines using Github (ensure that you create a private repo):
    1. Download the Github project as a zip (option on the main page of your Github repo)
    2. Rename the zip following the format mentioned in the point above, then send that zip to me by email.

If anything is unclear, please send me an email as soon as possible!

Grading Criteria:

Style/submission guidelines: https://gmierzwinski.github.io/bishops/cs403/final_project.html

Comments, Formatting, & Readability	**7.5%**
Submission Guidelines	**2.5%**
Robot AST	**20%**
Type Checking	**20%**
Scope Checking	**20%**
Code Execution	**10%**
Testing	**10%**
Report	**5%**
Creativity	**5%**
BONUS: Rover Army	**5%**
Total	**105%**