The program gets the command from a txt file. The program will pass that txt file through a parser, then it checks scope, then semantics, and then runs the commands that were sent. For entering the nodes of the parser: We tried really hard to not make it only one file, we were thinking about having a second file that would be sent the parsing code, which then would be accessed, parsed, ran and when a rover token reached would send it to the main rover file with a token to say it's a command. For two-way communication, the rover would itself write and the code would block until the value is read. we started implementing that, but it was really daunting and although quite in line with the project, calling it a satellite and making it as if the rover communicated through a satellite that does the brunt of the work and only sends the curated commands saving which in real life would save weight on the rover on computers and have better communication from the big antenna in space, but it was just too much.

Very sorry, there is definitively a better solution, but we can't think of any that are quick enough

to be implemented at the current time. The main problem we have is with global variables

that need to be accessed by reference which isn't possible by simple import as the file needs to be started. Instead, we tried to separate the code with this block of text and a bunch of end lines.

Robot movements:

* The robot takes the commands from the text file, parses them, checks scope then semantics, and then runs the program.
* The features are:
  + Moving forward, backward, left, and right relative to its orientation.
  + The robot is able to dig one square to the left, right, forward, or backward relative to the rover, and only if there is a treasure in one of these positions.
  + The robot can set a base in the rover position on the map and save its coordination.
  + The robot can retrieve all the positions of the bases that were set.
  + The robot is able to send info about its coordination, the map, all the treasures' positions on the map if it was taken or not, and the vehicle orientation.

For the grammar of the language, our program was inspired by assignment 4 that were sent before by the two of us and the language grammar is :

Create nodes + parse tree using grammar:

<program> ::= <block>

<block> ::= { <decls> <stmts> }

<decls> ::= e

| <decl> <decls>

<decl> ::= <type> ID ;

<type> ::= BASIC

<stmts> ::= e

| <stmt> <stmts>

<stmt> ::= <loc> = <bool> ;

| IF ( <bool> ) <stmt>

| IF ( <bool> ) <stmt> ELSE <stmt>

| WHILE ( <bool> ) <stmt>

| <block>

<loc> ::= ID

<bool> ::= <join> <boolcl>

<boolcl> ::= e

| || <join> <boolcl>

<join> ::= <equality> <joincl>

<joincl> ::= e

| && <equality> <joincl>

<equality> ::= <rel> <equalcl>

<equalcl> ::= e

| == <rel> <equalcl>

| != <rel> <equalcl>

<rel> ::= <expr> <reltail>

<reltail> ::= e

| <= <expr>

| >= <expr>

| > <expr>

| < <expr>

<expr> ::= <term> <exprcl>

<exprcl> ::= e

| + <term> <exprcl>

| - <term> <exprcl>

<term> ::= <unary> <termcl>

<termcl> ::= e

| \* <unary> <termcl>

| / <unary> <termcl>

<unary> ::= ! <unary>

| - <unary>

| <factor>

<factor> ::= ( <bool> )

| <loc>

| NUM

| REAL

| TRUE

| FALSE

"""

The parser works by chaining inside if statements verifications and tree building. We translate the rules by creating a node and passing it to a common verifier (this is the one from the superclass). Then the node is parsed inside the verifier and the return value is used to determine if the node is attached or not to the parent tree. this happens for each node recursively and when the parser arrives at a non-token node, it verifies if the name is legal and parses moves the cursor to the next token of code. this effectively makes this whole parser just a big if statement linked together and the final value of this of programNode is if the string is pared or not. An exception error can be created, but the output is still a bit buggy. Also note, that the string to be parsed has a first pass to tokenize the string between each space, then a program root node is created and parsed. The whole parser uses global variable for the cursor and parses the string for simplicity’s sake. All nodes are created from the same AbstractNode, so they have very similar functions.

“””

As we are a groupe of 2, we decided to make the scope more than 1 level and make bools and int interact and be able to set to each other.

For the scope, we have 3 levels, the current, the parent and the global. The current is the block that is currently ran and the parent being the parent of the block in which it resides. At the exit of the block the blockNode reassigns all the variables to their original values for those who were declared in the block. This makes it so that the scope is recursive and allows to have as many scopes as there is block like most programming languages we know of today. Finally, since the scope is empty at the beginning, it will also be empty at the end, therefore to interact with the rover, a global scope is used. This scope is a part of the rover class and allows the program to talk with the rover. All commands need to pass through the rover var (no need to declare, its in global scope) and assign it to the var with the method of the same id. For example:

rover = goRight ;

To get value from the environment, the language uses a way reminiscent of low level programming. To send an int to the program, one needs to set the global variable systemInt (once again no need to set as its global), then call the method the same way you would. For example:

systemInt = 5 ;

rover = setGround ;

To get a value, the same applies but the opposite:

rover = getGround ;

If ( systemInt == 5 )

I = 5 ;

For bools use instead systemBool

rover = canGoRight ;

if ( systemBool )

rover = goRight ;

The program also allows to print the system int, bool or the map with a R printed at the position of the rover.

The code allows you to store a value at the position, on the map, of the rover and get a value (see above examples). This would allow the rover to store data on the map and retrieve it for further use, such as a search algorithm based on a stack with values stored on a tile in binary.

To use the project, call the rover.py with a map path as an argument. Then call main.py with the path to the file containing the command string to be parsed. Only one arg per file is required, there is no two rover.