to design and implement a parser and interpreter for a simple programming language to control simple robots.

Important, part of this assignment will be auto marked. Please do not modify the template code's file names, class names and file names. In particular, DO NOT modify World.java file.

A variety of applications allow the user to "script" the application, or otherwise specify domain- specific programs to control, modify, or extend the application. Many advanced computer games have this facility, as do sophisticated editors of many kinds. All these applications will provide some kind of domain-specific language for specifying the scripts/programs, and must therefore also have a parser and interpreter to parse and execute the scripts.

In this assignment, your task will be to design and implement a parser and interpreter for a simple programming language that can be used to control robots for a simple robot game.

The RoboGame program is written already; your task is to add the parser and interpreter

We will provide a set of programs for testing each stage of your language interpreter.

Although it is not part of the assignment, you may wish to publish any robot programs you write on the forum so that other students can try running their robot programs against yours.

RoboGame is a program for a simple game involving two robots moving in a 2D grid based world that contains barrels of fuel.

The goal of the "game" is survival - the winner is the robot that still has fuel when the other one has run out.

* The robots start in opposite corners of the grid, and can move around the world, at each step moving forward one step, turning left, right or completely around, or remaining where it is.
* The robots require fuel, and use some up on every step. Their fuel level is displayed by a coloured arc that gets shorter as the fuel runs down. The robots stop, and the game ends, when the fuel level in one of them gets to zero.
* Barrels of fuel turn up at random places in the world. A robot that is on top of a barrel can take fuel from the barrel.
* A robot can also steal fuel from the other robot, if it is next to and facing the other, and the other robot doesn't have its shield up. Using the shield costs extra fuel.

The game has buttons for starting the game, and resetting the game to the start state.

It also has a menu for loading user programs into the robots.

If the robot has a loaded program, it will execute the program; otherwise, the robot performs a built-in default procedure, which constantly chases the closest barrel.

Your job is to write a parser and interpreter which can parse a robot program from a file and then execute it.

The robots can perform a variety of actions (move, turnL, etc),

and have sensors that return integer values specifying properties of the world (fuelLeft, wallDist etc).

The robot program language includes these actions and sensors

and also includes control structures (loops and conditionals) and operators for calculating and comparing.

The RoboGame programme consists of the following files:

RoboGame.java, with a main method which constructs the user interface.

WorldComponent.java, which manages the display of the state of the game.

World.java, which contains the code for simulating the world.

Robot.java, which contains the code for the individual robot objects. Your interpreter will call methods from the Robot class.

RobotProgramNode.java, which defines the type for the nodes in the abstract syntax tree that your parser will construct.

Each RobotProgramNode will have an execute method that takes a robot, and executes the program in the node on that robot.

Parser.java, which will contain your parser and interpreter. The very top level of the parser is already provided. The file also contains a main method that will help you test your parser quickly without having to run the whole RoboGame program.

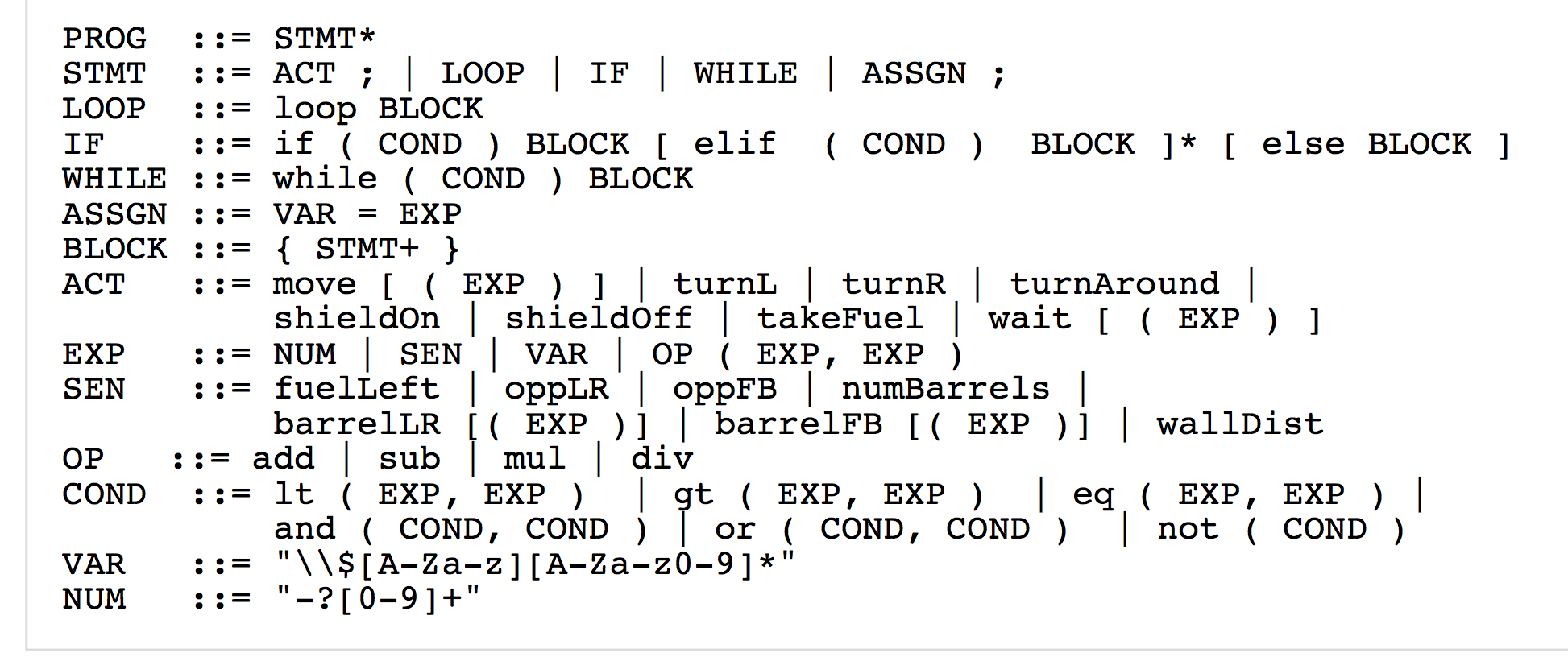
ParserFailureException and RobotInterruptedException, which declare exceptions used by the parser and robot simulator.

You must complete Parser.java by writing all the parse methods.

You must also define all the classes for the specific types of node, along with the methods in those classes (e.g. execute) that define the interpreter.

The full language (for stages 0 to 3) is specified in the following grammar, However, you should not attempt to build the parser for the whole language at once. The assignment lays out a sequence of increasing subsets of the language that you should progressively implement.

In this grammar (and later ones): Uppercase terms are NON-TERMINALS, lowercase and camel-case terms are terminals, [...] means optional, \* means zero or more occurrences of the preceding item, + means one or more occurrences of the preceding item, | is used to separate alternatives, ::= separates the left and right hand sides of a definition, and the definitions of VAR and NUM are regular expressions. All other symbols (brackets, commas and semicolons) are terminals.



None of the actions require arguments, but move and wait can take an optional argument.

The conditions in if or while statements can involve comparisons of integer valued expressions, or logical combinations of them using and, or, and not.

Expressions specifying values (EXP) can be sensor values, actual numbers, variables, or arithmetic expressions using add, sub, mul, or div.

Expressions are written in a prefix/functional form (e.g. eq(barrelFB, 0) or add(5,1)) for ease of parsing.

This will be replaced by infix expressions in the last part of the assignment.

Variables must start with a $, and can have numeric values assigned to them. The specification is a Java regular expression that matches variable names.

Numbers are integers, with an optional -ve sign. The specification is Java regular expression that matches numbers.

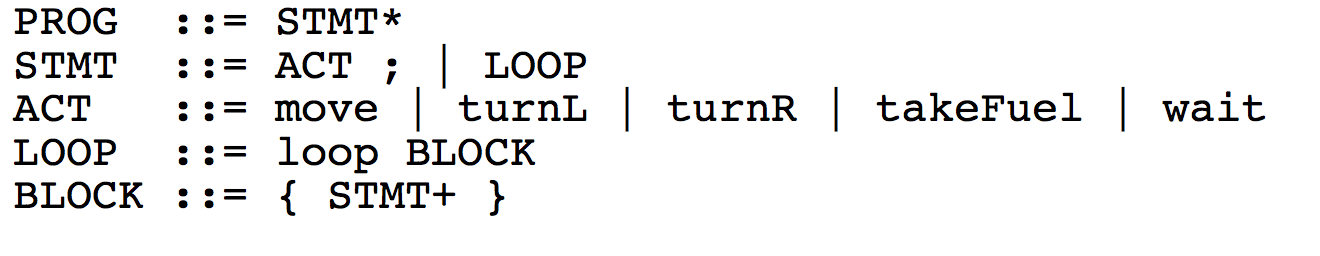
The sensors oppLR, oppFB, barrelLR, and barrelFB return the position of the opponent robot or the closest barrel, relative to the current position and direction of the robot. LR means the distance to the left (-ve) or right (+ve) , FB means the distance in front (+ve) or behind (-ve). If there are no barrels at present, then barrelLR and barrellFB will return a very large integer.

The sensors barrelLR, and barrelFB both take an optional argument, as in barrelLR(n) or barrelFB(n), where n specifies the nth closest barrel.

Any amount of white space (blanks, newlines and tabs) may occur between two adjacent terminals.

Stage 0: Getting started (40%)

For stage 0, you are to write a parser that can parse and execute a small subset of the language that has actions and loops without conditions, given by the follow grammar:



The following is an example program for this stage:

move; move; move; turnL ;

wait;

loop{

move; move; turnR;

move; move; turnR;

move; turnR;

move; move; turnR;

takeFuel;

}

You will need to define a node class for each of the non-terminals.

It is also sensible to define a node class for each of the actions.

Each node class should have an execute(Robot robot) method.

The execute methods for an action node class will call the relevant method from the Robot class on the given robot.

For example, for the TurnLNode class, it might be:

**public void** execute(Robot robot) {

robot.turnLeft();

}

Note that the method name in the Robot class is not necessarily the same name as the command in the robot language.

The execute method for LoopNode will not call methods on the robot directly,

but will repeatedly call the execute method of the BlockNode that it contains.

Similarly, the BlockNode will need to call the execute method of each of its components in turn.

The node classes should also have a toString method which returns a textual representation of the node.

The nodes corresponding to the PROG, STMT, LOOP, and BLOCK rules will need to construct the string out of their components.

For example, the LoopNode class might have the following method (assuming that block is a field containing the BlockNode that is contained in the LoopNode ):

**public** String toString() {

**return** "loop" + **this**.block;

}

You will also need to create a parse... method for each of the rules, which takes the scanner, and returns a RobotProgramNode.