Goal

Goal: Design and implement a language interpreter for a simple programming language that can be used to control simple robots.

To Submit

- The Java sources of the entire program, including your parser and the robot simulation code.
- Your `Parser.java` file (along with any additional files required by your parser).
- Your `report.txt` file

Introduction

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 language 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

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 world, and can move around the world, moving forward one step, turning left, right or completely around.
- 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 is over) 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. Without a program, a robot performs a built-in default procedure, which constantly chases the closest barrel. If the robot has a loaded program, it will execute the program.
Your job is to write the Parser and Interpreter which 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. `RobotProgramNode=s 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.

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 (eg `execute`) that define the interpreter.

The full language is specified in the following grammar, but you should not attempt to build the parser for the whole language at once - there is a sequence of increasing subsets of the language that you should progressively build.

Full grammar: [Uppercase terms are NON-TERMINALS; lowercase or camel-case terms are terminals.]

```
PROG  ::= STMT*

STMT  ::= ACT ; | LOOP | IF | WHILE | ASSGN ;

LOOP  ::= loop BLOCK

IF    ::= if ( COND ) BLOCK [ elif  ( COND )  BLOCK ]*
          [ else BLOCK ]

WHILE ::= while ( COND ) BLOCK

ASSGN ::= VAR = EXP

BLOCK ::= { STMT+ }

ACT   ::= move [ ( EXP ) ] | turnL | turnR |
          turnAround | shieldOn | shieldOff |
        takeFuel | wait [ ( EXP ) ]

EXP   ::= NUM | SEN | VAR | OP ( EXP, EXP )

SEN   ::= fuelLeft | oppLR | oppFB | numBarrels |
          barrelLR [( EXP )] | barrelFB [( EXP )] |
        wallDist

OP   ::= add | sub | mul | div

COND  ::= lt ( EXP, EXP )  | gt ( EXP, EXP )  |
          eq ( EXP, EXP ) | and ( COND, COND ) |
        or ( COND, COND )  | not ( COND )
```

```
VAR    ::= "\\$[A-Za-z][A-Za-z0-9]*"

NUM    ::= "-?[0-9]+"
```

Notes:

- None of the actions require arguments, but `move` and `wait` can take an optional argument.
- The conditions in the `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`.
- Variables must start with a $, and can have values assigned to them. The specification is in a java regular expression that will match variable names.
- Numbers are integers, with an optional -ve sign. The specification is java regular expression that will match 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 n'th closest barrel.

Here is a program in this language, along with some comments:

```
01 while(gt(fuelLeft, 0)){      // loop as long as fuel left is > 0
02     if(eq(numBarrels, 0)){   // if there are no barrels, then  wait
03         wait;
04     } elif ( lt(add(oppFB,oppLR), 3) ) {  //if opponent is close
05         move(oppFB);                       // (actually a wrong
calculation!)
06     } else{
07         $lr = barrelLR;       // put the relative position of
08         $fb = barrelFB;       // closest barrel into variables
09         if(and(eq($lr, 0), eq($fb, 0))){   // if robot is on top of a
barrel
10             takeFuel;                     // take the fuel
11         } else{
12             if(eq($fb, 0)){        // otherwise turn and move
13                 if(lt($lr, 0)){    //  towards the closest barrel
14                     turnL;
15                 } else{
16                     turnR;
17                 }
18             } else{
19                 if(gt($fb, 0)){
20                     move;
21                 } else{
22                     turnAround;
23                 }
24             }
25         }
26     }
27 }
```

Stage 0: Getting the parser going (40%)

For stage 0, write a parser that can parse and execute the following small subset of the language that has actions and loops without conditions.

```
PROG  ::= STMT*

STMT  ::= ACT ; | LOOP

ACT   ::= move | turnL | turnR | takeFuel | wait

LOOP  ::= loop BLOCK

BLOCK ::= { STMT+ }
```

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 node classes for each of the non-terminals. It is also wise to define node classes for each of the actions. The node classes should all have an `execute(Robot robot)` method. The `execute` methods of the action node classes will call the relevant method from the Robot class on the robot. For example, for the `TurnLNode` class, it might be:

```
1 public void execute(Robot robot){
2         robot.turnLeft();
3     }
```

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 in the `LoopNode` will not 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 that will return a description 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`):

```
1 public String toString(){
2         return "loop" + this.block;
3     }
```

You will also need to create `parse`... methods for each of the rules, which take the scanner, and return a RobotProgramNode.
Hint: There will be a lot of node classes. You can put each of them in a separate file, or you can include them all in the `Parser.java` file as non-public classes, since they are only accessed by the parser itself. It depends on your IDE which option is easier to handle.
Test your parser on the example program and the other test programs that we will provide.

1. Run the main method of the `Parser` class to check whether the parser parses programs correctly.
2. Once they parse correctly, run the `RoboGame` and load the programs into the robots to see whether the programs are executed correctly.

Stage 1 Basic language (up to 60%)

Extend your parser to handle the robot sensors and IF's and WHILE's. The conditions in the IF and WHILE statements can be restricted to simple comparisons of a sensor value with a number, eg `lt(fuelLeft, 20)` to determine whether there are less than 20 units of fuel left (the robot starts with 100 units).

```
PROG  ::= STMT*

STMT  ::= ACT; | LOOP |IF | WHILE

ACT   ::= move | turnL | turnR | turnAround | shieldOn |
          shieldOff | takeFuel | wait

LOOP  ::= loop BLOCK

IF    ::= if ( COND ) BLOCK

WHILE ::= while ( COND ) BLOCK

BLOCK ::= { STMT+ }

COND  ::= lt ( SEN, NUM )  | gt ( SEN, NUM )  |
          eq ( SEN, NUM )

SEN   ::= fuelLeft | oppLR | oppFB | numBarrels |
```

```
          barrelLR | barrelFB | wallDist
NUM   ::= "-?[0-9]+"
```

Here is an example program for this stage:

```
while ( gt(barrelFB, 0) ) { move; }
if (eq(barrelLR, 0)) {
   takeFuel;
}
if (lt(barrelLR, 0)) {
   turnL;
   while ( gt(barrelFB,0) ){ move;}
   takeFuel;
}
if (gt(barrelLR, 0)) {
   turnR;
   while ( gt(barrelFB,0) ){ move;}
   takeFuel;
}
wait;
loop {
  if ( gt(fuelLeft, 0) ) {
     move;
     turnL;
  }
}
```

You will need additional node classes and parse methods for the IF, WHILE, COND, and SEN rules. It is sensible to have classes for each of the comparisons (less than, greater than, and equal) and for each of the sensors (fuelLeft, etc). The `execute` methods for

the `IfNode` and `WhileNode` will need to perform the logic of testing the value of the condition in the node, and then executing the block in the node.
Note that the condition nodes (Cond, LessThan, etc) are a different type from the `RobotProgramNode` since they do not need an `execute` method but instead need an `evaluate` method, that will return a boolean value. You will need to define an interface type for this category of node. The `evaluate`method should have the robot as an argument, and should return a boolean.
The Sensor nodes are different again: like the condition nodes, they need an`evaluate` method, but their `evaluate` will return an `int` not a `boolean`. Their `evaluate` methods will need to call the appropriate methods on the robot: `getFuel()`, `getOpponentLR()`, `getOpponentFB()`, `numBarrels()`,`getClosestBarrelLR()` or `getClosestBarrelFB()`.

Stage 2: Arguments, Else, and Expressions (up to 75%)

Extend your parser to handle

- actions with optional arguments: `move` and `wait` can take an argument specifying how many move or wait steps to take.
- if statements with optional else clauses,
- arithmetic expressions that compute values with sensors and numbers
- more complex conditions with logical operators and expressions

```
PROG  ::= STMT*

STMT  ::= ACT; | LOOP | IF | WHILE

ACT   ::= move [ ( EXP ) ] | turnL | turnR |
          turnAround | shieldOn | shieldOff |
        takeFuel | wait [ ( EXP ) ]

LOOP  ::= loop BLOCK

IF    ::= if ( COND ) BLOCK [ else BLOCK ]

WHILE ::= while ( COND ) BLOCK

BLOCK ::= { STMT+ }

EXP   ::= NUM | SEN | OP ( EXP, EXP )

SEN   ::= fuelLeft | oppLR | oppFB | numBarrels |
        barrelLR | barrelFB | wallDist

OP    ::= add | sub | mul | div

COND  ::= and ( COND, COND ) | or ( COND, COND ) |
          not ( COND )  | lt ( EXP, EXP )  |
        gt ( EXP, EXP )  | eq ( EXP, EXP )
```

```
NUM   ::= "-?[1-9][0-9]*|0"
```

Here is a program for the stage 2 parser.

```
move(8);

turnL;

loop {

while ( or(eq(numBarrels, 0),

           lt(add(oppFB, oppLR), add(barrelFB,barrelLR)))
) {

  if (lt(oppFB,0)) { turnAround; }

  else { if (gt(oppFB,0)) { move(add(1, div(oppFB, 2)));
}

  else { if (lt(oppLR,0)) { turnL;}

  else { if (gt(oppLR,0)) { turnR;}

  else { if (eq(oppLR,0)) { takeFuel; }}}}}

}

if ( and(eq(barrelFB, 0),eq(barrelLR, 0))) { takeFuel; }

else { if ( lt(barrelFB, 0) ){ turnAround; }

else { if ( gt(barrelFB, 0) ) { move(barrelFB); }

else { if ( lt(barrelLR, 0) ) { turnL; }

else { if ( gt(barrelLR, 0) ) { turnR; }}}}}

}
```

You will need to

- add node classes and parse methods to handle the expressions.
- extend your parse methods for the if statement to handle an optional else. After parsing the condition and the "then" block, the method needs to check whether there is an "else" to determine whether it needs to parse an else block or simply return the IfNode without an else block. The `execute` method also needs to be extended.
- extend your parse methods for the move and wait actions to check for an optional argument. They should check for a "(" to determine whether there is an argument or not. The `execute` methods also need to be extended. Note that the Robot class does not provide a `move` or `idleWait` method with an argument - your `execute`method needs to call the `move` or `idlewait` method the specified number of times.

Stage 3 Variables (up to 85%)

Extend your parser to handle

- variables and assignment statements

- allow a sequence of elif elements in an if statement
- optional arguments to `barrelLR` and `barrelFB` to access the relative position of barrels other than the closest one.

```
PROG  ::= STMT*

STMT  ::= ACT ; | LOOP | IF | WHILE | ASSGN ;

LOOP  ::= loop BLOCK

IF    ::= if ( COND ) BLOCK [elif ( COND ) BLOCK]*
          [else BLOCK]

WHILE ::= while ( COND ) BLOCK

ASSGN ::= VAR = EXP

BLOCK ::= { STMT+ }

ACT   ::= move [( EXP )] | turnL | turnR | turnAround |
          shieldOn | shieldOff | takeFuel |
        wait [( EXP )]

EXP   ::= NUM | SEN | VAR | OP ( EXP, EXP )

SEN   ::= fuelLeft | oppLR | oppFB | numBarrels |
        barrelLR [( EXP )] | barrelFB [ ( EXP ) ] |
        wallDist

OP    ::= add | sub | mul | div

COND  ::= lt ( EXP, EXP )  | gt ( EXP, EXP )  |
          eq ( EXP, EXP ) | and ( COND, COND ) |
        or ( COND, COND ) | not ( COND )

NUM   ::= "-?[1-9][0-9]*|0"

VAR   ::= "\\$[A-Za-z][A-Za-z0-9]*"
```

The example program given near the beginning of this assignment is appropriate for stage 3.
Variables are identifiers starting with a $, and can hold integer values. Assignment statements can assign a value to a variable, and variables can be used inside expressions. Variables do not need to be declared. If they are used in an expression before a value has been assigned, then they are assumed to have the value 0. The scope of all variables is the whole program.

Evaluating an expression now needs to be able to access a map of all the current variable values, and an assignment statement needs to update the value of a variable in the map.
The `Robot` class provides four methods for accessing relative barrel position:`getClosestBarrelLR=(), =getClosestBarrelFB=(), =getBarrelLR(int n)` and `getClosestBarrelFB(int n)`. The last two return the relative position of the n'th closest barrel, allowing the program to identify barrels other than the closest one. With these, you could write robot programs that determine which barrel to aim for, if the opponent is already closer to the closest barrel.

Stage 4: Challenge (up to 100%)

Extend the language:

- allow infix operators and parentheses for both arithmetic and logic
- require variables to be declared before they can be used
- allow nested scope, so that variables declared inside a block (a) are only accessible within the block, and (b) "shadow" any variables of the same name declared in the program or outer blocks.

Implementing these extensions will require restructuring of the grammar in ways that have not been addressed in the lectures.

```
PROG  ::= STMT*

STMT  ::= ACT; | ASSGN; | LOOP | IF | WHILE | DO; |

          { STMT* }

ASSGN ::= VAR = EXP

VAR   ::= "\\$[A-Za-z][A-Za-z0-9]*"

LOOP  ::= loop STMT

IF    ::= if ( COND ) STMT [elif ( COND ) STMT ]*

          [else STMT]

WHILE ::= while ( COND ) STMT

DO    ::= do STMT while ( COND )

ACT   ::= move [( EXP )] | turnL | turnR |

        turnAround | shieldOn ( COND )  | takeFuel |

        wait [( EXP )]

EXP   ::= NUM | EXP OP EXP |  SEN | VAR | ( EXP )

SEN   ::= fuelLeft | oppLR | oppFB | numBarrels |

        barrelLR [( EXP )] | barrelFB [( EXP )] |

        wallDist
```

```
OP    ::= + | - | * | /

COND  ::= BOOL | COND LOGIC COND | ! COND |

          EXP COMP EXP | ( COND )

LOGIC ::= && | || | ^

COMP  ::= < | <= | > | >= | == | !=

NUM   ::= "-?[1-9][0-9]*|0"

BOOL  ::= true | false
```

What to hand in: code and report.

Submit

- your `Parser.java` file and any additional files you required in your implementation if you chose to place them in separate files.
- a jar file of the complete program so that it can be run by the marker.
- A brief report which lists the parts that you did. Include any of your own robot programs that you tested your parser on.