KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS

Programavimo kalbų teorija (P175B124)

Laboratorinių darbų ataskaita

Atliko:
IFF-1/6 gr. studentas
Lukas Kuzmickas
2023 m. kovo 26 d.
Priėmė:
Doc. Sajavičius Svajūnas

KAUNAS 2023

TURINYS

1. C++ arba Ruby (L1)

1.1. Darbo užduotis

You are to determinate X by given Y, from expression $X = \sqrt{Y}$

Input

The first line is the number of test cases, followed by a blank line.

Each test case of the input contains a positive integer Y ($1 \leq Y \leq 10^{1000}$), with no blanks or leading zeroes in it.

It is guaranteed, that for given Y, X will be always an integer.

Each test case will be separated by a single line.

Output

For each test case, your program should print X in the same format as Y was given in input.

Print a blank line between the outputs for two consecutive test cases.

Sample Input

```
1

7206604678144
```

Sample Output

```
2684512
```

1 pav. Užduoties pavyzdys iš rinkinių.

Užduotis labai paprasta, turime kintamuosius Y ir X. Turime įvedimo ir išvedimo failus, Y saugomas įvedimo faile, X išvedamas į išvedimo failą. Mes paskaičiuojame X reikšmę, pagal formulę $X = \sqrt{Y}$. Pradžioje įvedimo failo nurodome atvejų kiekį ir pačias Y reikšmes. X išvedamas su eilučių tarpais, jeigu turime daugiau nei vieną duomenų atvejį.

1.2. Programos tekstas

Program.cpp

```
//AUTHOR Lukas Kuzmickas IFF-1/6
#include <iostream>
#include <fstream>
#include <string>
#include <chrono>
#include <vector>
#include <iostream>
#include <sstream>
using namespace std;

/// <summary>
/// Input class for file input operations
/// </summary>
static class Input {
public:
    Input(const string& filename) : infile(filename) {}
    /// <summary>
    /// Method to read from string filename to an integer data type
    /// </summary>
    /// <returns>integer</returns>
    int readInt() {
        int n;
        infile >> n;
        return n;
    }

    /// <summary>
    /// Method to read a file to a string
    /// </summary>
    /// <returns>string</returns>
    string readString() {
        string s;
        infile >> s;
        return s;
    }

    /// <summary>
    /// Method to check if file isn't empty (End of file)
    /// </summary>
    /// <returns>true or false</returns>
    bool eof() {
        return infile.eof();
    }

private:
    ifstream infile;
};


/// <summary>
/// Output class for file output operations
/// </summary>
static class Output {
public:
    Output(const string& filename) : outfile(filename) {}

    /// <summary>
    /// Write function for writing output to a data file
    /// </summary>
```

```
    /// <param name="s">string to write</param>
    void write(const string& s) {
        outfile << s;
    }

    /// <summary>
    /// Write function for writing output with new line to a data file
    /// </summary>
    /// <param name="s">string value</param>
    void writeLine(const string& s) {
        outfile << s << endl;
    }
private:
    ofstream outfile;
};


/// <summary>
/// Data class for Y object
/// </summary>
class Y {
private:
    string y_str;
public:
    Y(const string& str) : y_str(str) {}
    /// <summary>
    /// Putting all Y value to a vector digit list
    /// </summary>
    /// <returns>vector value digit list</returns>
    vector<int> digits() const {
        vector<int> y_digits(y_str.size());
        for (int i = 0; i < y_str.size(); i++) {
            //ASCI code convertion/deletion
            y_digits[i] = y_str[i] - '0';
        }
        return y_digits;
    }
};

/// <summary>
/// Data class for X object
/// </summary>
class X {
public:
    X(int value) : value(value) {}
    /// <summary>
    /// Convert X value to string
    /// </summary>
    /// <returns>string</returns>
    string to_string() const {
        return std::to_string(value);
    }
private:
    int value;
};


/// <summary>
/// Class for data operations
/// </summary>
class TaskUtils {
public:
    /// <summary>
    /// Method for converting from string to double
    /// </summary>
    /// <param name="str">given string</param>
```

```
    /// <returns>double value</returns>
    static double stringToDouble(const std::string& str) {
        double result;
        std::istringstream stream(str);
        stream >> result;
        return result;
    }
    /// <summary>
    /// Method for getting root square of given value
    /// </summary>
    /// <param name="Y">Given value</param>
    /// <returns>root square</returns>
    static double squared(double Y)
    {
        return sqrt(Y);
    }
};
int main() {
    //INPUT for Y
    Input in("input_y.txt");
    //OUTPUT for X
    Output out("output_x.txt");
    using namespace std::chrono;
    //chrono objects for time calculation
    time_point<system_clock> start, end;
    start = system_clock::now();

    //number of test cases
    int t = in.readInt();
    while (!in.eof() && t > 0) {
        Y y(in.readString());
        vector<int> y_digits = y.digits();
        string y_str;
        for (int i = 0; i < y_digits.size(); i++) {
            y_str += to_string(y_digits[i]);
        }
        double Y_double = TaskUtils::stringToDouble(y_str);
        X x = TaskUtils::squared(Y_double);
        out.writeLine(x.to_string());
        if (!in.eof()) {
            out.writeLine("");
        }
        t--;
    }
    end = system_clock::now();
    duration<double> elapsed_seconds = end - start;
    cout << "Total time elapsed " << elapsed_seconds.count() << "s\n";
    return 0;
}
```

1.3. Pradiniai duomenys ir rezultatai

Pirmieji duomenys:

input_y.txt
```
2
169
81
```

output_x.txt
```
13

9
```

Antrieji duomenys:

input_y.txt
```
10
9000000
31382404
40401
324
9
25
16
4
81
64
```
output_x.txt
```
3000

5602

201

18

3

5

4

2

9

8
```

2. Scala L2 darbas

2.1. Darbo užduotis

Turime sukurti Scalatron botą, naudodami Scala programavimo kalbą, pagal pateiktus reikalavimus.

Reikalavimai programai/botui

1. Panaudoti bent kelis master boto išleidžiamus botų padėjėjų tipus
(pvz.: minos, raketos į priešus, "kamikadzės", rinkikai, masalas ir pan.)
2. Panaudoti bet kurį vieną iš kelio radimo algoritmų (DFS, BFS, A*, Greedy, Dijkstra).

2 pav. Reikalavimai botui.

2.2. Programos tekstas

```
//Lukas Kuzmickas IFF-1/6
import scala.collection.mutable.Stack
import scala.util.Random



//Pasyvus botas, resursų rinkėjas su galimybėmis dėti minas ir bombas. Naudoja DFS
algoritmą resursų rinkimui.

/** This bot builds a 'direction value map' that assigns an attractiveness score
to
  * each of the eight available 45-degree directions. Additional behaviors:
  * - aggressive missiles: approach an enemy master, then explode
  * - defensive missiles: approach an enemy slave and annihilate it
  *
  * The gatherers (mini-bots) use DFS algorithm to find shortest path to collect a
resource
  *
  * The master bot uses the following state parameters:
  *  - dontFireAggressiveMissileUntil
  *  - dontFireDefensiveMissileUntil
  *  - lastDirection
  * The mini-bots use the following state parameters:
  *  - mood = Aggressive | Defensive | Gathering | Mine | Returning | Bomb
  *  - target = remaining offset to target location
  */
object ControlFunction
{
    def forMaster(bot: Bot) = {
        val random = new Random()
        val (directionValues, nearestEnemyMaster, nearestEnemySlave,
numberOfResources, nearbyEnemies) = analyzeViewAsMaster(bot.view)
        val dontFireAggressiveMissileUntil =
bot.inputAsIntOrElse("dontFireAggressiveMissileUntil", -1)
        val dontFireDefensiveMissileUntil =
bot.inputAsIntOrElse("dontFireDefensiveMissileUntil", -1)
        val lastDirection = bot.inputAsIntOrElse("lastDirection", 0)
        // determine movement direction
        directionValues(lastDirection) += 10 // try to break ties by favoring the
last direction
        val bestDirection45 = directionValues.zipWithIndex.maxBy(_._1)._2
        val direction = XY.fromDirection45(bestDirection45)
        bot.move(direction)
```

```
        bot.set("lastDirection" -> bestDirection45)

        if(dontFireAggressiveMissileUntil < bot.time && bot.energy > 100) { // 
fire attack missile?
            nearestEnemyMaster match {
                case None =>            // no-on nearby
                case Some(relPos) =>    // a master is nearby
                    val unitDelta = relPos.signum
                    val remainder = relPos - unitDelta // we place slave nearer 
target, so subtract that from overall delta
                    bot.spawn(unitDelta, "mood" -> "Aggressive", "target" -> 
remainder)
                    bot.set("dontFireAggressiveMissileUntil" -> (bot.time + 
relPos.stepCount + 1))
            }
        }
        else
        if(dontFireDefensiveMissileUntil < bot.time && bot.energy > 100) { // fire 
defensive missile?
            nearestEnemySlave match {
                case None =>            // no-on nearby
                case Some(relPos) =>    // an enemy slave is nearby
                    if(relPos.stepCount < 8) {
                        // this one's getting too close!
                        val unitDelta = relPos.signum
                        val remainder = relPos - unitDelta // we place slave 
nearer target, so subtract that from overall delta
                        bot.spawn(unitDelta, "mood" -> "Defensive", "target" -> 
remainder)
                        bot.set("dontFireDefensiveMissileUntil" -> (bot.time + 
relPos.stepCount + 1))
                    }
            }
        }
        if(bot.time < 3900 && numberOfResources > 1){ //gatherer bot
            for(i <- 0 until numberOfResources.min(7)){
                if(bot.energy > 200){
                    val spawnLoc = XY.fromDirection45(i)
                    bot.spawn(spawnLoc, "mood" -> "Gathering")
                }
            }
        }
        else if(nearbyEnemies > 3 && bot.energy >= 300) {
            bot.spawn(XY(random.nextInt(3)-1, random.nextInt(3)-1), "mood" -> 
"Mine")
        }


        //Check if bot needs bombs
       else if (bot.energy > 500 && nearbyEnemies > 5) {
           bot.spawn(XY(random.nextInt(3)-1, random.nextInt(3)-1), "mood" -> 
"Bomb")
           bot.say("I spawned a bomb!")
        }
    }



    //Describes all moods of a mini bot
    def forSlave(bot: MiniBot) = {
        bot.inputOrElse("mood", "Idle") match {
            case "Aggressive" => reactAsAggressiveMissile(bot)
            case "Defensive" => reactAsDefensiveMissile(bot)
            case "Gathering" => reactAsGathering(bot)
            case "Returning" => reactAsReturning(bot)
            case "Bomb" => reactAsBomb(bot)
```

```
            case "Mine" => reactAsMine(bot)
            case s: String => bot.log("unknown mood: " + s)
        }
    }




    /**
      * Describes the behaviour of a bomb
      */
    def reactAsBomb(bot: MiniBot): Unit = {
        bot.explode(8)
    }


    /**
      * Describes the behaviour of a aggressive missile bot
      */
    def reactAsAggressiveMissile(bot: MiniBot) = {
        bot.view.offsetToNearest('m') match {
            case Some(delta: XY) =>
                // another master is visible at the given relative position (i.e.
position delta)
                // close enough to blow it up?
                if(delta.length <= 2) {
                    // yes -- blow it up!
                    bot.explode(4)
                } else {
                    // no -- move closer!
                    bot.move(delta.signum)
                    bot.set("rx" -> delta.x, "ry" -> delta.y)
                }
            case None =>
                // no target visible -- follow our targeting strategy
                val target = bot.inputAsXYOrElse("target", XY.Zero)
                // did we arrive at the target?
                if(target.isNonZero) {
                    // no -- keep going
                    val  unitDelta  =  target.signum  //  e.g.  CellPos(-8,6)  =>
CellPos(-1,1)
                    bot.move(unitDelta)
                    //  compute  the  remaining  delta  and  encode  it  into  a  new
'target' property
                    val remainder = target - unitDelta // e.g. = CellPos(-7,5)
                    bot.set("target" -> remainder)
                } else {
                    // yes -- but we did not detonate yet, and are not pursuing
anything?!? => switch purpose
                    bot.set("mood" -> "Gathering", "target" -> "")
                    bot.say("Gathering")
                }
        }
    }








    /**
      * Describes the behavior of a defensive missile
      */
    def reactAsDefensiveMissile(bot: MiniBot) = {
```

```
        bot.view.offsetToNearest('s') match {
            case Some(delta: XY) =>
                // another slave is visible at the given relative position (i.e.
position delta)
                // move closer!
                bot.move(delta.signum)
                bot.set("rx" -> delta.x, "ry" -> delta.y)
            case None =>
                // no target visible -- follow our targeting strategy
                val target = bot.inputAsXYOrElse("target", XY.Zero)
                // did we arrive at the target?
                if(target.isNonZero) {
                    // no -- keep going
                    val unitDelta = target.signum // e.g. CellPos(-8,6) =>
CellPos(-1,1)
                    bot.move(unitDelta)
                    // compute the remaining delta and encode it into a new
'target' property
                    val remainder = target - unitDelta // e.g. = CellPos(-7,5)
                    bot.set("target" -> remainder)
                } else {
                    // yes -- but we did not annihilate yet, and are not pursuing
anything?!? => switch purpose
                    bot.set("mood" -> "Gathering", "target" -> "")
                    bot.say("Gathering")
                }
        }
    }



    /**
      * Describes the behaviour of a mine bot
      */
    def reactAsMine(bot: MiniBot) {
        var blewUp = false
        val targets = Array('s', 'b', 'm')
        for (t <- targets) {
            bot.view.offsetToNearest(t) match {
                case Some(delta: XY) =>
                    // close enough to blow it up?
                    if(delta.length <= 2 && !blewUp) {
                        // yes -- blow it up
                        bot.say("Boom")
                        bot.explode(4)
                        blewUp = true
                    }
                case None => // do nothing
            }
        }
    }





    /**
      * Describes the behaviour of a gatherer slave bot
      */
    def reactAsGathering(bot: MiniBot) = {
        if(bot.time > 3900){
            //deposit energy to master
            bot.set("mood" -> "Returning")
        }
        else{
```

```
            val bestDirectionToFood = findDirectionToGatherDFS(bot.view) //DFS for
direction of movement
            if(bestDirectionToFood == -1){ // not path to food
                if(bot.energy < 2000){
                    bot.set("mood" -> "Returning")
                }
            }
            else{ //bot finds path to food
                val direction = XY.fromDirection45(bestDirectionToFood)
                bot.move(direction)
                bot.set("lastDirection" -> bestDirectionToFood)
            }
        }

    }




    /**
      * Defines the behaviour of a bot that's returning to master
      */
    def reactAsReturning(bot: MiniBot) = {
        // determine movement direction towards master
        val (directionValues) = analyzeViewAsReturning(bot.view) // other
directions
        directionValues(bot.offsetToMaster.toDirection45) += 400 // direction to
master
        val bestDirection45 = directionValues.zipWithIndex.maxBy(_._1)._2
        val direction = XY.fromDirection45(bestDirection45)
        bot.move(direction)
        bot.set("lastDirection" -> bestDirection45)
    }





    /** Analyze the view, building a map of attractiveness for the 45-degree
directions and
      * recording other relevant data, such as the nearest elements of various
kinds.
      */
    def analyzeViewAsMaster(view: View) = {
        val directionValues = Array.ofDim[Double](8) // slightly confusing, but
this is a 1 dimension array of length 8 (number of directions)
        var nearestEnemyMaster: Option[XY] = None // coordinates of the nearest
enemy master ?
        var nearestEnemySlave: Option[XY] = None // coordinates of the nearest
enemy slave ?
        var nearbyEnemies = 0.0
        var numberOfResources: Int = 0
        val cells = view.cells // takes the string of cells contained in the
master's view
        val cellCount = cells.length // determines the length of the cells string
(number of cells seen by the master)
        for(i <- 0 until cellCount) { // iterates through the cells
            val cellRelPos = view.relPosFromIndex(i) // gets the relative position
of a certain cell
            if(cellRelPos.isNonZero) {
                val stepDistance = cellRelPos.stepCount // how many steps it takes
to get to a cell
                val value: Double = cells(i) match { // calculates the value of
every cell
                    case 'm' => // another master: not dangerous, but an obstacle
                        nearbyEnemies += 1
```

```
                        nearestEnemyMaster = Some(cellRelPos)
                        if(stepDistance < 2) -1000 else 0
                    case 's' => // another slave: potentially dangerous?
                        nearbyEnemies += 1
                        nearestEnemySlave = Some(cellRelPos)
                        -100 / stepDistance
                    case 'S' => // our own slave
                        0.0
                    case 'B' => // good beast: valuable, but runs away
                        numberOfResources = numberOfResources + 1
                        if(stepDistance == 1) 600
                        else if(stepDistance == 2) 300
                        else (150 - stepDistance * 15).max(10)
                    case 'P' => // good plant: less valuable, but does not run
                        numberOfResources = numberOfResources + 1
                        if(stepDistance == 1) 500
                        else if(stepDistance == 2) 300
                        else (150 - stepDistance * 10).max(10)
                    case 'b' => // bad beast: dangerous, but only if very close
                        nearbyEnemies += 1
                        if(stepDistance < 4) -400 / stepDistance else -50 /
stepDistance
                    case 'p' => // bad plant: bad, but only if I step on it
                        if(stepDistance < 2) -1000 else 0
                    case 'W' => // wall: harmless, just don't walk into it
                        if(stepDistance < 2) -1000 else 0
                    case _ => 0.0
                }
                val direction45 = cellRelPos.toDirection45
                directionValues(direction45) += value
            }
        }
        (directionValues, nearestEnemyMaster, nearestEnemySlave,
numberOfResources, nearbyEnemies)
    }




    /**
      * Analyzes view as a bot that means to return to master
      */
    def analyzeViewAsReturning(view: View) = {
        val directionValues = Array.ofDim[Double](8)
        val cells = view.cells // the view of the gatherer
        val cellCount = cells.length
        for(i <- 0 until cellCount) { // iterates through the cells
            val cellRelPos = view.relPosFromIndex(i) // gets the relative position
of a certain cell
            if(cellRelPos.isNonZero) { // checks if the position is not center
(aka the gatherer bot)
                val stepDistance = cellRelPos.stepCount // how many steps it takes
to get to a cell
                val value: Double = cells(i) match { // calculates the value of
every cell
                    case 'M' => // our master, very good as we want to deposit
                        1000
                    case 'm' => // different master: potentially dangerous?
                        if(stepDistance < 2) -1000 else -100 + (-100 /
stepDistance)
                    case 's' => // another slave: potentially dangerous?
                        -100 + (-100 / stepDistance)
                    case 'S' => // other slave: : harmless, just don't walk into
it
                        if(stepDistance < 2) -1000 else 0
```

```
                    case 'B' => // good beast but we want to move towards master
so lesser priority
                        if(stepDistance == 1) 600
                        else if(stepDistance == 2) 100
                        else 0
                    case 'P' => // good plant: even less valuable to us
                        if(stepDistance == 1) 600
                        else if(stepDistance == 2) 300
                        else 0
                    case 'b' => // bad beast: dangerous, but only if very close
                        if(stepDistance < 4) -400 / stepDistance else -50 /
stepDistance
                    case 'p' => // bad plant: bad, but only if I step on it
                        if(stepDistance < 2) -1000 else 0
                    case 'W' => // wall: harmless, just don't walk into it
                        if(stepDistance < 2) -1000 else 0
                    case _ => 0.0
                }
                val direction45 = cellRelPos.toDirection45
                directionValues(direction45) += value
            }
        }
        directionValues
    }


/**
  * Uses the DFS path finding algorithm to determine which direction the bot
should move
  * to go down the path towards the closest food resource
  */
  def findDirectionToGatherDFS(view: View): Int = {
    val centerCell = view.center
    var cellStack = List(centerCell)
    var visited = Set(centerCell)
    var initialDirection = Map(centerCell -> -1) // the center cell has no initial
direction
    var pathDirection = -1 // sets the initial direction

  while (cellStack.nonEmpty && pathDirection == -1) {
    val currentCell = cellStack.head
    cellStack = cellStack.tail

    if (!visited.contains(currentCell)) {
      visited += currentCell
      val cellElement = view.cells(view.indexFromAbsPos(currentCell))
      if ("BP".contains(cellElement)) {
        val cellDirection = initialDirection(currentCell)
        pathDirection = cellDirection
      }
      else {
        val neighbors = getNeighbors(currentCell, view)
        for (neighbor <- neighbors) {
          if (!visited.contains(neighbor)) {
            cellStack = neighbor :: cellStack
            if (!initialDirection.contains(currentCell)) {
              initialDirection += neighbor ->
view.relPosFromAbsPos(neighbor).toDirection45
            }
            else {
              initialDirection += neighbor -> initialDirection(currentCell)
            }
          }
        }
      }
    }
```

```
    }

    pathDirection
  }

  // Help function that returns all neighbors of a cell
  def getNeighbors(cell: XY, view: View): Seq[XY] = {
    val  neighbors  =  Seq(cell  +  XY.Right,  cell  +  XY.Up,  cell  +  XY.Left,  cell  +
XY.Down)
    val validNeighbors = for (neighbor <- neighbors
                              if (neighbor.x >= 0 && neighbor.x < view.size &&
                                neighbor.y >= 0 && neighbor.y < view.size &&

!"msSbpW".contains(view.cells(view.indexFromAbsPos(neighbor)))))
                          yield neighbor
    validNeighbors
  }


}
```

```
// -----------------------------------------------------------------------------
------------------
// Framework
// -----------------------------------------------------------------------------
------------------
class ControlFunctionFactory {
    def create = (input: String) => {
        val (opcode, params) = CommandParser(input)
        opcode match {
            case "React" =>
                val bot = new BotImpl(params)
                if( bot.generation == 0 ) {
                    ControlFunction.forMaster(bot)
                } else {
                    ControlFunction.forSlave(bot)
                }
                bot.toString
            case _ => "" // OK
        }
    }
}
// -----------------------------------------------------------------------------
------------------
trait Bot {
    // inputs
    def inputOrElse(key: String, fallback: String): String
    def inputAsIntOrElse(key: String, fallback: Int): Int
    def inputAsXYOrElse(keyPrefix: String, fallback: XY): XY
    def view: View
    def energy: Int
    def time: Int
    def generation: Int
    // outputs
    def move(delta: XY) : Bot
    def say(text: String) : Bot
    def status(text: String) : Bot
    def spawn(offset: XY, params: (String,Any)*) : Bot
    def set(params: (String,Any)*) : Bot
    def log(text: String) : Bot
}
```

```
trait MiniBot extends Bot {
    // inputs
    def offsetToMaster: XY // minibots also know the direction towards master
(used by gatherers)
    // outputs
    def explode(blastRadius: Int) : Bot // and minibots can explode
}
case class BotImpl(inputParams: Map[String, String]) extends MiniBot {
    // input
    def inputOrElse(key: String, fallback: String) = inputParams.getOrElse(key,
fallback)
    def inputAsIntOrElse(key: String, fallback: Int) =
inputParams.get(key).map(_.toInt).getOrElse(fallback)
    def inputAsXYOrElse(key: String, fallback: XY) = inputParams.get(key).map(s =>
XY(s)).getOrElse(fallback)
    val view = View(inputParams("view"))
    val energy = inputParams("energy").toInt
    val time = inputParams("time").toInt
    val generation = inputParams("generation").toInt
    def offsetToMaster = inputAsXYOrElse("master", XY.Zero)
    // output
    private var stateParams = Map.empty[String,Any]     // holds "Set()" commands
    private var commands = ""                               // holds all other
commands
    private var debugOutput = ""                           // holds all "Log()"
output
    /** Appends a new command to the command string; returns 'this' for fluent
API. */
    private def append(s: String) : Bot = { commands += (if(commands.isEmpty) s
else "|" + s); this }
    /** Renders commands and stateParams into a control function return string. */
    override def toString = {
        var result = commands
        if(!stateParams.isEmpty) {
            if(!result.isEmpty) result += "|"
            result += stateParams.map(e => e._1 + "=" +
e._2).mkString("Set(",",",")")
        }
        if(!debugOutput.isEmpty) {
            if(!result.isEmpty) result += "|"
            result += "Log(text=" + debugOutput + ")"
        }
        result
    }
    def log(text: String) = { debugOutput += text + "\n"; this }
    def move(direction: XY) = append("Move(direction=" + direction + ")")
    def say(text: String) = append("Say(text=" + text + ")")
    def status(text: String) = append("Status(text=" + text + ")")
    def explode(blastRadius: Int) = append("Explode(size=" + blastRadius + ")")
    def spawn(offset: XY, params: (String,Any)*) =
        append("Spawn(direction=" + offset +
            (if(params.isEmpty) "" else "," + params.map(e => e._1 + "=" +
e._2).mkString(",")) +
            ")")
    def set(params: (String,Any)*) = { stateParams ++= params; this }
    def set(keyPrefix: String, xy: XY) = { stateParams ++= List(keyPrefix+"x" ->
xy.x, keyPrefix+"y" -> xy.y); this }
}
// -------------------------------------------------------------------------------
------------------
/** Utility methods for parsing strings containing a single command of the format
  * "Command(key=value,key=value,...)"
  */
object CommandParser {
    /** "Command(..)" => ("Command", Map( ("key" -> "value"), ("key" -> "value"),
..}) */
```

```
    def apply(command: String): (String, Map[String, String]) = {
        /** "key=value" => ("key","value") */
        def splitParameterIntoKeyValue(param: String): (String, String) = {
            val segments = param.split('=')
            (segments(0), if(segments.length>=2) segments(1) else "")
        }
        val segments = command.split('(')
        if( segments.length != 2 )
            throw new IllegalStateException("invalid command: " + command)
        val opcode = segments(0)
        val params = segments(1).dropRight(1).split(',')
        val keyValuePairs = params.map(splitParameterIntoKeyValue).toMap
        (opcode, keyValuePairs)
    }
}
// ---------------------------------------------------------------------------------
------------------
/** Utility class for managing 2D cell coordinates.
  * The coordinate (0,0) corresponds to the top-left corner of the arena on
screen.
  * The direction (1,-1) points right and up.
  */
case class XY(x: Int, y: Int) {
    override def toString = x + ":" + y
    def isNonZero = x != 0 || y != 0
    def isZero = x == 0 && y == 0
    def isNonNegative = x >= 0 && y >= 0
    def updateX(newX: Int) = XY(newX, y)
    def updateY(newY: Int) = XY(x, newY)
    def addToX(dx: Int) = XY(x + dx, y)
    def addToY(dy: Int) = XY(x, y + dy)
    def +(pos: XY) = XY(x + pos.x, y + pos.y)
    def -(pos: XY) = XY(x - pos.x, y - pos.y)
    def *(factor: Double) = XY((x * factor).intValue, (y * factor).intValue)
    def distanceTo(pos: XY): Double = (this - pos).length // Phythagorean
    def length: Double = math.sqrt(x * x + y * y) // Phythagorean
    def stepsTo(pos: XY): Int = (this - pos).stepCount // steps to reach pos: max
delta X or Y
    def stepCount: Int = x.abs.max(y.abs) // steps from (0,0) to get here: max X
or Y
    def signum = XY(x.signum, y.signum)
    def negate = XY(-x, -y)
    def negateX = XY(-x, y)
    def negateY = XY(x, -y)
    /** Returns the direction index with 'Right' being index 0, then clockwise in
45 degree steps. */
    def toDirection45: Int = {
        val unit = signum
        unit.x match {
            case -1 =>
                unit.y match {
                    case -1 =>
                        if(x < y * 3) Direction45.Left
                        else if(y < x * 3) Direction45.Up
                        else Direction45.UpLeft
                    case 0 =>
                        Direction45.Left
                    case 1 =>
                        if(-x > y * 3) Direction45.Left
                        else if(y > -x * 3) Direction45.Down
                        else Direction45.LeftDown
                }
            case 0 =>
                unit.y match {
                    case 1 => Direction45.Down
```

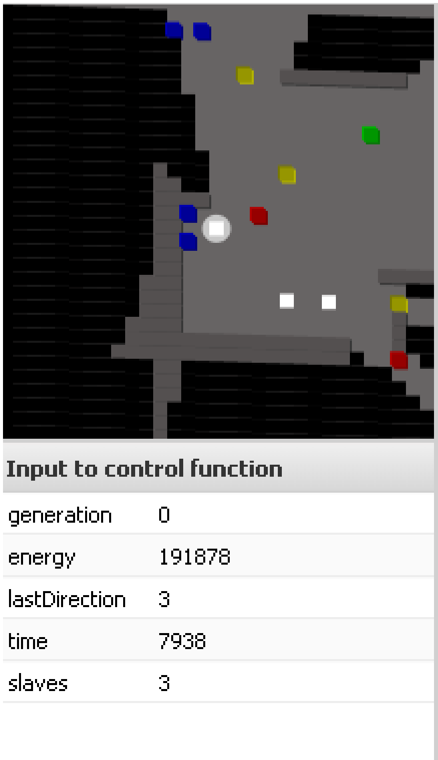
```
                    case 0 => throw new IllegalArgumentException("cannot compute
direction index for (0,0)")
                    case -1 => Direction45.Up
                }
            case 1 =>
                unit.y match {
                    case -1 =>
                        if(x > -y * 3) Direction45.Right
                        else if(-y > x * 3) Direction45.Up
                        else Direction45.RightUp
                    case 0 =>
                        Direction45.Right
                    case 1 =>
                        if(x > y * 3) Direction45.Right
                        else if(y > x * 3) Direction45.Down
                        else Direction45.DownRight
                }
        }
    }
    def rotateCounterClockwise45 = XY.fromDirection45((signum.toDirection45 + 1) %
8)
    def rotateCounterClockwise90 = XY.fromDirection45((signum.toDirection45 + 2) %
8)
    def rotateClockwise45 = XY.fromDirection45((signum.toDirection45 + 7) % 8)
    def rotateClockwise90 = XY.fromDirection45((signum.toDirection45 + 6) % 8)
    def wrap(boardSize: XY) = {
        val fixedX = if(x < 0) boardSize.x + x else if(x >= boardSize.x) x -
boardSize.x else x
        val fixedY = if(y < 0) boardSize.y + y else if(y >= boardSize.y) y -
boardSize.y else y
        if(fixedX != x || fixedY != y) XY(fixedX, fixedY) else this
    }
}
object XY {
    /** Parse an XY value from XY.toString format, e.g. "2:3". */
    def apply(s: String) : XY = { val a = s.split(':'); XY(a(0).toInt,a(1).toInt)
}
    val Zero = XY(0, 0)
    val One = XY(1, 1)
    val Right     = XY( 1,  0)
    val RightUp   = XY( 1, -1)
    val Up        = XY( 0, -1)
    val UpLeft    = XY(-1, -1)
    val Left      = XY(-1,  0)
    val LeftDown  = XY(-1,  1)
    val Down      = XY( 0,  1)
    val DownRight = XY( 1,  1)
    def fromDirection45(index: Int): XY = index match {
        case Direction45.Right => Right
        case Direction45.RightUp => RightUp
        case Direction45.Up => Up
        case Direction45.UpLeft => UpLeft
        case Direction45.Left => Left
        case Direction45.LeftDown => LeftDown
        case Direction45.Down => Down
        case Direction45.DownRight => DownRight
    }
    def fromDirection90(index: Int): XY = index match {
        case Direction90.Right => Right
        case Direction90.Up => Up
        case Direction90.Left => Left
        case Direction90.Down => Down
    }
    def apply(array: Array[Int]): XY = XY(array(0), array(1))
}
object Direction45 {
```

```
    val Right = 0
    val RightUp = 1
    val Up = 2
    val UpLeft = 3
    val Left = 4
    val LeftDown = 5
    val Down = 6
    val DownRight = 7
}
object Direction90 {
    val Right = 0
    val Up = 1
    val Left = 2
    val Down = 3
}
// -------------------------------------------------------------------------------
------------------
case class View(cells: String) {
    val size = math.sqrt(cells.length).toInt
    val center = XY(size / 2, size / 2)
    def apply(relPos: XY) = cellAtRelPos(relPos)
    def indexFromAbsPos(absPos: XY) = absPos.x + absPos.y * size
    def absPosFromIndex(index: Int) = XY(index % size, index / size)
    def absPosFromRelPos(relPos: XY) = relPos + center
    def cellAtAbsPos(absPos: XY) = cells.charAt(indexFromAbsPos(absPos))
    def indexFromRelPos(relPos: XY) = indexFromAbsPos(absPosFromRelPos(relPos))
    def relPosFromAbsPos(absPos: XY) = absPos - center
    def relPosFromIndex(index: Int) = relPosFromAbsPos(absPosFromIndex(index))
    def cellAtRelPos(relPos: XY) = cells.charAt(indexFromRelPos(relPos))
    def offsetToNearest(c: Char) = {
        val matchingXY = cells.view.zipWithIndex.filter(_._1 == c)
        if( matchingXY.isEmpty )
            None
        else {
            val          nearest          =          matchingXY.map(p          =>
relPosFromIndex(p._2)).minBy(_.length)
            Some(nearest)
        }
    }
}
```

2.3. Pradiniai duomenys ir rezultatai

Aprašome mūsų botą ir galime paleisti jį į turnyrą prieš kitą botą.

4 pav. Mūsų boto turnyras prieš reference botą.



5 pav. Botas-rinkėjas.

3. L3

3.1. Darbo užduotis

3.2. Programos tekstas

3.3. Pradiniai duomenys ir rezultatai

4. L4

4.1. Darbo užduotis

4.2. Programos tekstas

4.3. Pradiniai duomenys ir rezultatai