KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS

Programavimo kalbų teorija (P175B124)

Laboratorinių darbų ataskaita

Atliko:
IFF-6/11 gr. studentas
Nerijus Dulkė
2019 m. birželio 6 d.
Priėmė:
lekt. **Evaldas Guogis**

KAUNAS 2019

TURINYS

1. Python (L1)

1.1. *Darbo užduotis*

Nuoroda į užduotį:

https://uva.onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=9&page=show_problem&problem=725

Trumpas aprašymas:

Labirintas sudarytas iš stačiakampių kambarių pavaizduotas plokštumoje, naudojant simbolius. Užduoties tikslas yra pažymėti kambarius, kuriuos galima aplankyti iš nurodytos startinės pozicijos.

```
XXXXXXXXXXXXXXXXXXXXX
X   X   X   X   X   X
X           X   X   X
X   X   X   X   X   X
XXXXXX XXX XXXXXXXXXX
X   X   X   X   X   X
X   X     *         X
X   X   X   X   X   X
XXXXXXXXXXXXXXXXXXXXX
```

a) Initial maze

```
XXXXXXXXXXXXXXXXXXXXX
X###X###X###X   X   X
X###########X   X   X
X###X###X###X   X   X
XXXXXX#XXX#XXXXXXXXXX
X   X###X###X###X###X
X   X###############X
X   X###X###X###X###X
XXXXXXXXXXXXXXXXXXXXX
```

b) Painted maze

1.2. *Programos tekstas*

Nerijus.Dulke.IFF.6.11.Lab.1.py

```python
# IFF-6/11 Nerijus Dulke Lab1
#
https://uva.onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=9&page
=show_problem&problem=725

from Maze import Maze

mazes = []

duom = open('duom.txt')

mazeCount = int(duom.readline())

i = 0
iterations = 0
while i < mazeCount:
  maze = Maze()
  line = duom.readline()

  while line and not line.startswith('_'):
    maze.addline(line)
    iterations = iterations + 1
    line = duom.readline()

  mazes.append(maze)
  i = i + 1

duom.close()

rez = open('rez.txt', 'w+')
```

```
for maze in mazes:
  maze.paint()
  maze.printresult(rez)
  rez.write('_____\n')

rez.close()
```

Maze.py

```
WALL = 'X'
EMPTY = ' '
MARKED = '#'
START = '*'
directions = ['u', 'd', 'l', 'r', 'ul', 'ur', 'dl', 'dr']

class Maze:
  def __init__(self):
    self.lines = []
    self.startX = -1
    self.startY = -1
    self.maxX = -1
    self.maxY = -1

  def addline(self, line):
    self.lines.append(line)

    if self.startX == -1:
      index = line.find(START)
      if index != -1:
        self.startX = index
        self.startY = self.count() - 1

    self.maxY = self.count() - 1
    maxX = len(line) - 1
    if maxX > self.maxX:
      self.maxX = maxX
    return

  def printlines(self):
    for line in self.lines:
      print line
    return

  def count(self):
    return len(self.lines)

  def printresult(self, file):
    file.writelines(self.lines)
    return

  def getvalue(self, coord):
    return self.lines[coord.y][coord.x]
```

```
  def markvalue(self, coord):
    if self.getvalue(coord) is EMPTY:
      self.lines[coord.y] = self.lines[coord.y][:coord.x] + MARKED +
self.lines[coord.y][coord.x + 1:]
      return True
    return False

  def paint(self):
    self.max = Coord(self.maxX, self.maxY)
    current = Coord(self.startX, self.startY)
    coordsToSearch = [current]
    visited = []

    while len(coordsToSearch) > 0:
      current = Coord(coordsToSearch[0].x, coordsToSearch[0].y)
      del coordsToSearch[0]

      for direction in directions:
        neighbour = self.getNeighbour(current, direction)
        isVisited = filter(lambda x: neighbour.equals(x), visited)

        if neighbour is None or len(isVisited) > 0:
          continue

        success = self.markvalue(neighbour)
        if success:
          coordsToSearch.append(neighbour)

      visited.append(current)
    return

  def getNeighbour(self, current, direction):
    coord = Coord(current.x, current.y)
    if not coord.canmove(direction, self.max):
      return None

    if direction is 'u':
      coord.up()
    elif direction is 'd':
      coord.down()
    elif direction is 'l':
      coord.left()
    elif direction is 'r':
      coord.right()
    elif direction is 'ul':
      coord.up()
      coord.left()
    elif direction is 'ur':
      coord.up()
      coord.right()
    elif direction is 'dl':
      coord.down()
      coord.left()
```

```
    elif direction is 'dr':
      coord.down()
      coord.right()

    return coord


class Coord:
  def __init__(self, x, y):
    self.x = x
    self.y = y

  def canmove(self, direction, maxcoord):
    if direction is 'u':
      return self.y != 0
    elif direction is 'd':
      return self.y != maxcoord.y
    elif direction is 'l':
      return self.x != 0
    elif direction is 'r':
      return self.x != maxcoord.x
    elif direction is 'ul':
      return self.y != 0 and self.x != 0
    elif direction is 'ur':
      return self.y != 0 and self.x != maxcoord.x
    elif direction is 'dl':
      return self.y != maxcoord.y and self.x != 0
    elif direction is 'dr':
      return self.y != maxcoord.y and self.x != maxcoord.x
    return False

  def up(self):
    self.y = self.y - 1

  def down(self):
    self.y = self.y + 1

  def left(self):
    self.x = self.x - 1

  def right(self):
    self.x = self.x + 1

  def equals(self, other):
    return self.x == other.x and self.y == other.y
```

1.3. Pradiniai duomenys ir rezultatai

duom.txt

```
2
XXXXXXXXX
X   X   X
X *     X
X   X   X
XXXXXXXXX
X   X
X   X
X   X
XXXXX
_____
XXXXX
X   X
X * X
X   X
XXXXX
_____
```

rez.txt

```
XXXXXXXXX
X###X###X
X#*#####X
X###X###X
XXXXXXXXX
X   X
X   X
X   X
XXXXX
_____
XXXXX
X###X
X#*#X
X###X
XXXXX
_____
```

2. Scala (L2)

2.1. Užduotis

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

2.2. Programos tekstas

```
// Example Bot #1: The Reference Bot

object ControlFunction
{
  def forMaster(bot: Bot) {
    val (directionValue, nearestEnemyMaster, nearestEnemySlave, nearestFood) = analyzeViewAsMaster(bot.view)

    val dontFireAggressiveMissileUntil = bot.inputAsIntOrElse("dontFireAggressiveMissileUntil", -1)
    val dontFireDefensiveMissileUntil = bot.inputAsIntOrElse("dontFireDefensiveMissileUntil", -1)
    val lastDirection = bot.inputAsIntOrElse("lastDirection", 0)
```

```
        // determine movement direction
        directionValue(lastDirection) += 10 // try to break ties by favoring the last direction
        val bestDirection45 = directionValue.zipWithIndex.maxBy(_._1)._2
        val direction = XY.fromDirection45(bestDirection45)
        bot.move(direction)
        bot.set("lastDirection" -> bestDirection45)

        if(bot.energy > -1000){
            nearestFood match{
                case None =>
                case Some(relPos) =>
                    val unitDelta = relPos.signum
                    val remainder = relPos - unitDelta
                    bot.spawn(unitDelta, "mood" -> "Picker", "target" -> remainder)
            }
        }
        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))
                    }
            }
        }
    }

    def forSlave(bot: MiniBot) {
        bot.inputOrElse("mood", "Lurking") match {
            case "Aggressive" => reactAsAggressiveMissile(bot)
            case "Defensive" => reactAsDefensiveMissile(bot)
            case "Picker"=> reactAsPicker(bot)
            case "Kamikaze" => reactAsKamikaze(bot)
            case s: String => bot.log("unknown mood: " + s)
        }
    }

    def reactAsPicker(bot: MiniBot){
        val (directionValue, nearestEnemyMaster, nearestEnemySlave, nearestFood) =
analyzeViewAsMaster(bot.view)
            bot.view.offsetToNearest('p') match {
```

```
        case Some(delta: XY) =>
          {
            bot.move(delta.signum)
            bot.set("rx" -> delta.x, "ry" -> delta.y)
          }
        case None =>
          val target = bot.inputAsXYOrElse("target", XY.Zero)

          // have reached target?
          if(target.isNonZero) {
            val unitDelta = target.signum
            bot.move(unitDelta)

            val remainder = target - unitDelta
            bot.set("target" -> remainder)
          } else {
            val bestDirection45 = directionValue.zipWithIndex.maxBy(_._1)._2
            val direction = XY.fromDirection45(bestDirection45)
            bot.move(direction)
          }
      }
    }

    def reactAsKamikaze(bot: MiniBot) {
      val (directionValue, nearestEnemyMaster) = analyzeViewAsKamikaze(bot, bot.view)
      if (nearestEnemyMaster.isDefined && nearestEnemyMaster.get.stepCount <= 2) {
        bot.say("Kamikaze")
        bot.explode(4)
        return
      }

      val lastDirection = bot.inputAsIntOrElse("lastDirection", 0)
      directionValue(lastDirection) += 10
      val bestDirection45 = directionValue.zipWithIndex.maxBy(_._1)._2
      val direction = XY.fromDirection45(bestDirection45)
      bot.move(direction)
      bot.set("lastDirection" -> bestDirection45)
    }

    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" -> "Lurking", "target" -> "")
                bot.say("Lurking")
            }
    }
}

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" -> "Lurking", "target" -> "")
                bot.say("Lurking")
            }
    }
}

/** 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 directionValue = Array.ofDim[Double](8)
    var nearestEnemyMaster: Option[XY] = None
    var nearestEnemySlave: Option[XY] = None
    var nearestFood: Option[XY] = None

    val cells = view.cells
    val cellCount = cells.length
    for(i <- 0 until cellCount) {
        val cellRelPos = view.relPosFromIndex(i)
        if(cellRelPos.isNonZero) {
```

```
            val stepDistance = cellRelPos.stepCount
            val value: Double = cells(i) match {
                case 'm' => // another master: not dangerous, but an obstacle
                    nearestEnemyMaster = Some(cellRelPos)
                    if(stepDistance < 2) -1000 else 0

                case 's' => // another slave: potentially dangerous?
                    nearestEnemySlave = Some(cellRelPos)
                    -100 / stepDistance

                case 'S' => // out own slave
                    0.0

                case 'B' => // good beast: valuable, but runs away
                    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
                    nearestFood = Some(cellRelPos)
                    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
                    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
            directionValue(direction45) += value
        }
    }
    (directionValue, nearestEnemyMaster, nearestEnemySlave, nearestFood)
}

def analyzeViewAsKamikaze(bot: MiniBot, view: View) = {
    val directionValue = Array.ofDim[Double](8)
    var nearestEnemyMaster: Option[XY] = None

    val cells = view.cells
    val cellCount = cells.length
    for(i <- 0 until cellCount) {
        val cellRelPos = view.relPosFromIndex(i)
        if(cellRelPos.isNonZero) {
            val stepDistance = cellRelPos.stepCount
            val value: Double = cells(i) match {
                case 'm' => // another master: not dangerous, but an obstacle
                    nearestEnemyMaster = Some(cellRelPos)
                    -100 / stepDistance

                case 'b' => // bad beast: dangerous, but only if very close
                    if(stepDistance < 4) -400 / stepDistance else -50 / stepDistance
```

```
                case 'W' => // wall: harmless, just don't walk into it
                    if(stepDistance < 2) -1000 else 0

                case _ => 0.0
            }
            val direction45 = cellRelPos.toDirection45
            directionValue(direction45) += value
        }
    }
    (directionValue, nearestEnemyMaster)
  }
}

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

    // outputs
    def explode(blastRadius: Int) : Bot
}
```

```
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 {
    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) = {
```

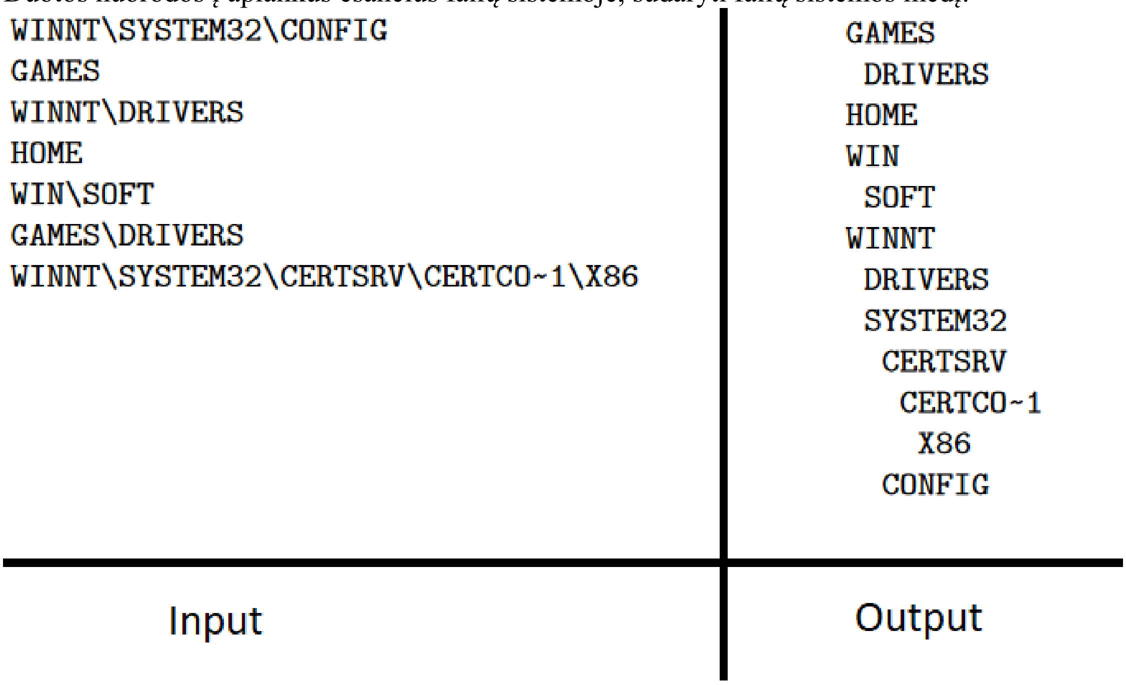
```
    val matchingXY = cells.view.zipWithIndex.filter(_._1 == c)
    if( matchingXY.isEmpty )
      None
    else {
      val nearest = matchingXY.map(p => relPosFromIndex(p._2)).minBy(_.length)
      Some(nearest)
    }
  }
}
```

3. F# (L3)

3.1. Darbo užduotis

Nuoroda į užduotį:
https://uva.onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=448&page=show_problem&problem=4331

Trumpas aprašymas:
Duotos nuorodos į aplankus esančius failų sistemoje, sudaryti failų sistemos medį.



3.2. Programos tekstas

```
//
https://uva.onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=448&page=show_problem&problem=4331

open System
open System.IO

let readLines filePath = File.ReadLines(filePath)

let print (line: string) = printf "%s\n" line

let startsWithCount (path: string, lines: seq<string>) =
    lines
    |> Seq.filter(fun (y: string) -> y.StartsWith(path))
    |> Seq.length

let replaceFirst (text: string, search: string, replace: string) =
    let pos = text.IndexOf(search)
    if pos < 0 then
        text
    else
```

```
        text.Substring(0, pos) + replace + text.Substring(pos + search.Length)

let hasDepth (lines: seq<string>) = Seq.exists(fun x -> String.exists(fun c -> c.Equals '\\')
x) lines

let rec handleThings (lines: seq<string>, spaces: string) : (seq<string>) =
    if hasDepth lines then
        let transformedLines =
            lines
            |> Seq.filter(fun x -> startsWithCount(x, lines).Equals 1)
            |> Seq.map(fun x -> replaceFirst(x, "\\", "\n" + spaces))

        handleThings(transformedLines, (spaces + "  "))
    else
        lines

[<EntryPoint>]
let main argv =
    let lines = readLines "data.txt" |> Seq.sort
    let result =
        handleThings(lines, "  ")
        |> Seq.map(fun x -> x.Split('\n'))
        |> Seq.concat
        |> Seq.distinct

    Seq.iter print result

    Console.ReadKey() |> ignore
    0 // return an integer exit code
```

3.3. Pradiniai duomenys ir rezultatai

duom.txt

```
WINNT\SYSTEM32\CONFIG
GAMES
HOME
WIN\SOFT
GAMES\DRIVERS
WINNT\SYSTEM32\CERTSRV\CERTCO~1\X86
```

Rezultatai

```
GAMES
  DRIVERS
HOME
WINNT
  SYSTEM32
    CERTSRV
      CERTCO~1
        X86
    CONFIG
WIN
  SOFT
```

4. Prolog (L4)

4.1. Darbo užduotis

Gautos užduotys iš sąrašo – 8 ir 9:

8. Rekursiškai suskaičiuokite sąrašų (bet kokio gylio) sveikų skaičių sumą

9. Dviejų dimensijų sąraše raskite palindromus (žodžiai, iš abiejų pusių skaitomi vienodai)

4.2. Programos tekstas

```
uzd8([], Sum) :- format("~a~n", Sum).
uzd8([H|T], Sum) :-
    integer(H),
    NewSum is Sum + H,
    uzd8(T, NewSum)
    ;
    not(integer(H)),
    uzd8(T, Sum).

isReverse(List) :-
    reverse(List, List),
    string_codes(Str, List),
    format("~a~n", Str).
isReverse(_).

row([]).
row([H|T]) :-
    string_codes(H, Chars),
    isReverse(Chars),
    row(T).

uzd9([]).
uzd9([H|T]) :-
    row(H),
    uzd9(T).

start :-
    writeln('8 uzd atsakymas:'),
    Numbers = [1, 2, 4, 1.3, 4.5, 3],
    uzd8(Numbers, 0),
    writeln('9 uzd atsakymas:'),
    Words = [["aba", "bbb", "ca"],["ds","eegee","fa"]],
    uzd9(Words).
```

4.3. Pradiniai duomenys ir rezultatai

8 užduoties duomenys: [1, 2, 4, 1.3, 4.5, 3]

8 užduoties rezultatai: 10

9 užduoties duomenys:
["aba", "bbb", "ca"],
["ds","eegee","fa"]

9 užduoties rezultatai:
aba
bbb
eegee