COMP3000_Programming Languages

Assignment 1: Solve Snake Puzzle

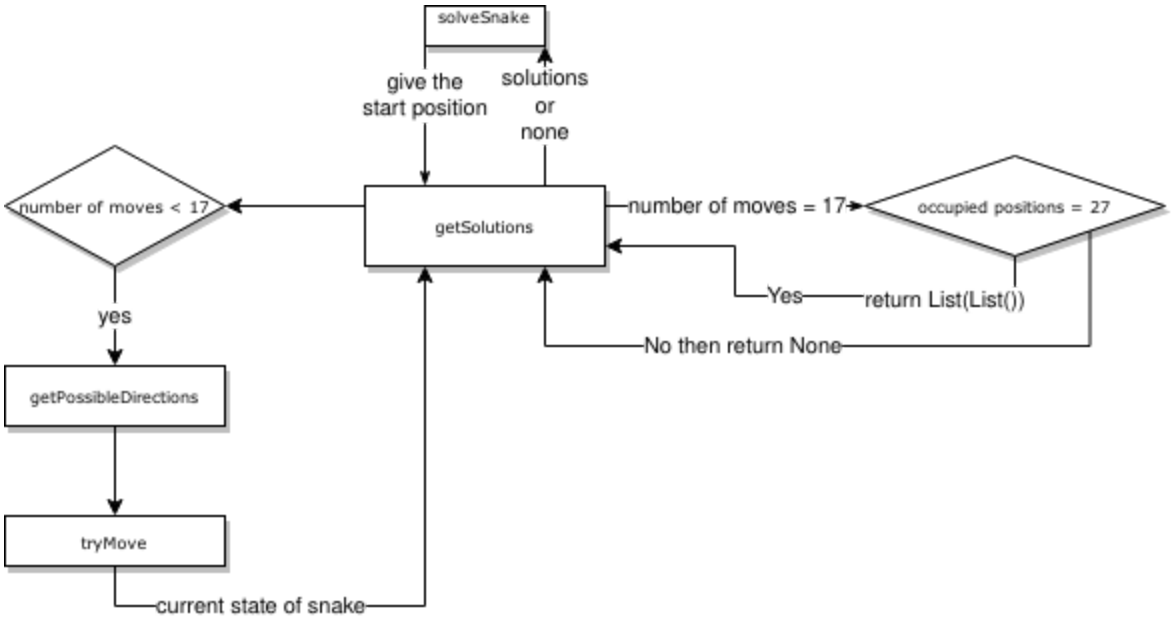
Cong Trinh

SID: 44580460

Introduction

The goal of the assignment is to introduce a functional programming language Scala. In this assignment, solve the snake puzzle is used for this assignment due to the nature of the puzzle that is suitable for functional programming.

Design



Implementation

Functions	Description
getMove	Input: move: Int, dir: Direction Output: (Int, Int, Int) This function is to calculate the number of units that the snake segment will fold. For example, XPos with 3 produce (3,0,0)
getOccupied	Input: state: CubeState, old: (Int, Int, Int), newPosition: (Int, Int, Int), dir: Direction Output: Set[(Int, Int, Int)] To calculate the set of new occupied positions from the current position to the new position. It separates two states: StartState and OtherState. With the StartState, the current position, which is the start position will be added to the occupied list. `case StartState(segments, start) => {` `  if (old != newPosition)` `  Set(old) ++ getOccupied(state, newPos, newPosition, dir)` `  else Set(newPosition)` `}` However, with the OtherState, the current position is already added to the occupied list with the previous calculation. `case OtherState(segments, occupied, end, direction, numMoves) => {` `  if (newPos != newPosition)` `  Set(newPos) ++ getOccupied(state, newPos, newPosition, dir)` `  else Set(newPos)` `}`
isValid	Input: x: (Int, Int, Int) Output: Boolean It takes the position and checks if the position is within the boundaries of the 3x3 cube.
addList	Input: x: (Int, Int, In), y: (Int, Int, Int) Output: (Int, Int, Int) It takes 2 tuples of 3 integers and produces the new tuple of the sum of the 2 tuples.
tryMove	Input: state: CubeState, move: Direction

	Output: Option[CubeState] The function will take the current state of the snake and a direction that the function will test if the next move is possible. The first step is to calculate a new position from the given direction. Next to check the new position is valid by using isValid function. The last thing the function check is if the new occupied positions produced by getOccupied is intersected with the current state's occupied list. There is one thing that I notice is for the StartState, I have to subtract 1 for the new position because of the positions are calculated from 0 to 2 instead 1 to 3.
getSolutions	Input: state: CubeState Output: Option[List[List[Direction]] It is coded to produce all possible solutions to the given state of the snake by recursively trying all the possibilities directions for each move. Each possible direction is computed by calling getPossibleDirections and is filtered by function tryMove. The first check is for StartState and OtherState. The StartState only check XPos direction. For the OtherState, I check numMoves as following: 1) If the numMoves equals 1, test if YPos is possible due to specification 2) If the numMoves equals segments list size (in default case is 17): a) If all the positions are occupied (occupied.size == 27), return Some(List(List())) b) Otherwise return None 3) If the numMoves is less than the segments list size: get all the possible directions for that state, then compute the valid directions by running tryMove function with all the possible directions. There are 2 versions of code that I developed. The first one (version 1) is `validDirections match {` `case x if x.size == 1 => {` `Some(prependDir(x(0),` `getSolutions(tryMove(state,` `x(0)).get).getOrElse((List(List()))))` `}` `case x if x.size == 2 => {` `Some(prependDir(x(0),` `getSolutions(tryMove(state,` `x(0)).get).getOrElse((List(List()))))` `++ prependDir(x(1), getSolutions(tryMove(state,` `x(1)).get).getOrElse((List(List()))))` `)` `}` `case x if x.size == 3 => {`

```
            Some(prependDir(x(0),
getSolutions(tryMove(state,
x(0)).get).getOrElse((List(List()))))
            ++ prependDir(x(1), getSolutions(tryMove(state,
x(1)).get).getOrElse((List(List()))))
            ++ prependDir(x(2), getSolutions(tryMove(state,
x(2)).get).getOrElse((List(List()))))
            )
          }
          case x if x.size == 4 => {
            Some(prependDir(x(0),
getSolutions(tryMove(state,
x(0)).get).getOrElse((List(List()))))
            ++ prependDir(x(1), getSolutions(tryMove(state,
x(1)).get).getOrElse((List(List()))))
            ++ prependDir(x(2), getSolutions(tryMove(state,
x(2)).get).getOrElse((List(List()))))
            ++ prependDir(x(3), getSolutions(tryMove(state,
x(3)).get).getOrElse((List(List()))))
          }
          case x if x.size == 0 => {
            None
          }
        }
```

Note: validDirections is the filtered of possible directions after being processed by tryMove function.

The version 1 is working fine and produce the correct list of direction at the end, however, for failure result, instead of producing None, it produces an empty list. Since the specification of the assignment states that the function must return failure, I modify the code into version 2, which is more concise and satisfy the specification.

```
val solutions = validDirections.map(direction => {
            getSolutions(tryMove(state, direction).get)
match {
              case None      => List(List())
              case Some(sol) => prependDir(direction, sol)
            }
          }).filter(x => {x != List(List())})
          println(solutions)
```

	`` if (solutions == List()) None else Some(solutions.flatten) ```

Test cases

Function	Test	
prependDir	```prependDir(XPos, List(List())) == List(List(XPos))```	The function should be able to add direction to an empty list.
prependStart	```prependStart((0,0,0), List(List())) == List(((0,0,0),List()))```	The function should be able to add direction to an empty list.
	``prependStart((3,0,0), List(List())) == List()```	This tests if the function still adds invalid start position (i.e. out of boundaries of 3x3 cube). Ideally, it should return None but because of its configuration, I only can return an empty list.
getIncrement	I do not have more test for this function.	
getPossibleDirections	```getPossibleDirections(st6).toSet.equals(Set(XPos))```	Even though I did not use this function to get startState's direction but it should be able to produce XPos when needed.
tryMove	```tryMove(st6, YPos) == Some(OtherState(List(3, 1,```	Even though the function is given a YPos, it still treats StartState with XPos.

	`1, 2, 1, 2, 1, 1, 2, 2, 1, 1, 1, 2, 2, 2, 2),Set((0,0,0), (1,0,0), (2,0,0)),(2,0,0),XPos,1))`	
	`tryMove(st3, XPos) == None`	The function should not accept the fully occupied cube since the getSolution will check the condition. However, the function should check in case the getSolutions runs improperly.
getSolutions		
	`getSolutions(st1) == None` `getSolutions(st2) == None`	Both of these do not yield solutions because the branches from these do not contain the final solutions for the given snake (checked by solving by hand)
	`getSolutions(st6) == Some(List(List(XPos, YPos, XNeg, ZPos, YPos, ZNeg, XPos, ZPos, YNeg, XNeg, YPos, ZNeg, YPos, ZPos, YNeg, XPos, YPos)))`	It should produce the solution because it is the solution to the given snake.
	`getSolutions(st10) == Some(List(List(XPos, YPos, XNeg, ZPos, XPos, ZNeg, XPos, YNeg, XNeg, ZPos, YPos, ZNeg, XPos, ZPos, YNeg, XPos, YPos)))`	Test the function with secondary snake (snake02). It produces confirmed solution (by hand)
	`getSolutions(` `StartState(snake02,(0,1,0)`	Check if the function works with snake02.

	`)` `) == None`	
getOccupied		
	`!getOccupied(st6, (0,0,0), (2,0,0), XPos).isEmpty`	Check if the function works with StartState.
	`!getOccupied(st5, (0,0,0), (2,0,0), XPos).isEmpty`	Check if the function works with OtherState. The function should work regardless of the occupied positions of the current state. It only produces the list of positions that are between the current position and new position.
	`!getOccupied(st5, (0,0,0), (-1,0,0), XPos).isEmpty`	It does not allow an invalid new position. It should return an empty set. Even though the tryMove function already tests if the new position is valid or not, this function will double-check.
isValid		
	`isValid(2,2,2) == true`	Check if valid position returns true.
	`isValid(-1,-1,2) == false`	Check if invalid position return false
addList		
	`addList((1,3,4),(3,4,5)) == (4,7,9)`	Check if the function can add two positive numbers.
	`addList((-1,3,-4),(3,4,5)) == (2,7,1)`	Check if the function can add positive and negative numbers.
	`addList((-1,-1,-1),(-2,-2,-2)) == (-3,-3,-3)`	Check if it can add two negative numbers.

solveSnake	

```
solveSnake(snake) ==
List(((0,0,0),List(XPos,
YPos, XNeg, ZPos, YPos,
ZNeg, XPos, ZPos, YNeg,
XNeg, YPos, ZNeg, YPos,
ZPos, YNeg, XPos, YPos)))

solveSnake(snake02) ==
List(((0,0,0),List(XPos,
YPos, XNeg, ZPos, XPos,
ZNeg, XPos, YNeg, XNeg,
ZPos, YPos, ZNeg, XPos,
ZPos, YNeg, XPos, YPos)))
```

I have not found the snake that gives two solutions within the specification of the assignment. However, these solutions are confirmed by hand.

```
solveSnake(snake) ==
List(((0,0,0),List(XPos,
YPos, XNeg, ZPos, XPos,
ZNeg, XPos, YNeg, XNeg,
ZPos, YPos, ZNeg, XPos,
ZPos, YNeg, XPos, YPos)),
((0,0,2),List(XPos, YPos,
XNeg, ZNeg, XPos, ZPos,
XPos, YNeg, XNeg, ZNeg,
YPos, ZPos, XPos, ZNeg,
YNeg, XPos, YPos)))
```

I only can test this if I remove the symmetrical restrictions from the assignment specification.