Authors:
Vsemirnov Roman
Liadov Lev
Polupanova Anna
Sirotkina Veronika
Trofimova Kristina

1. Introduction

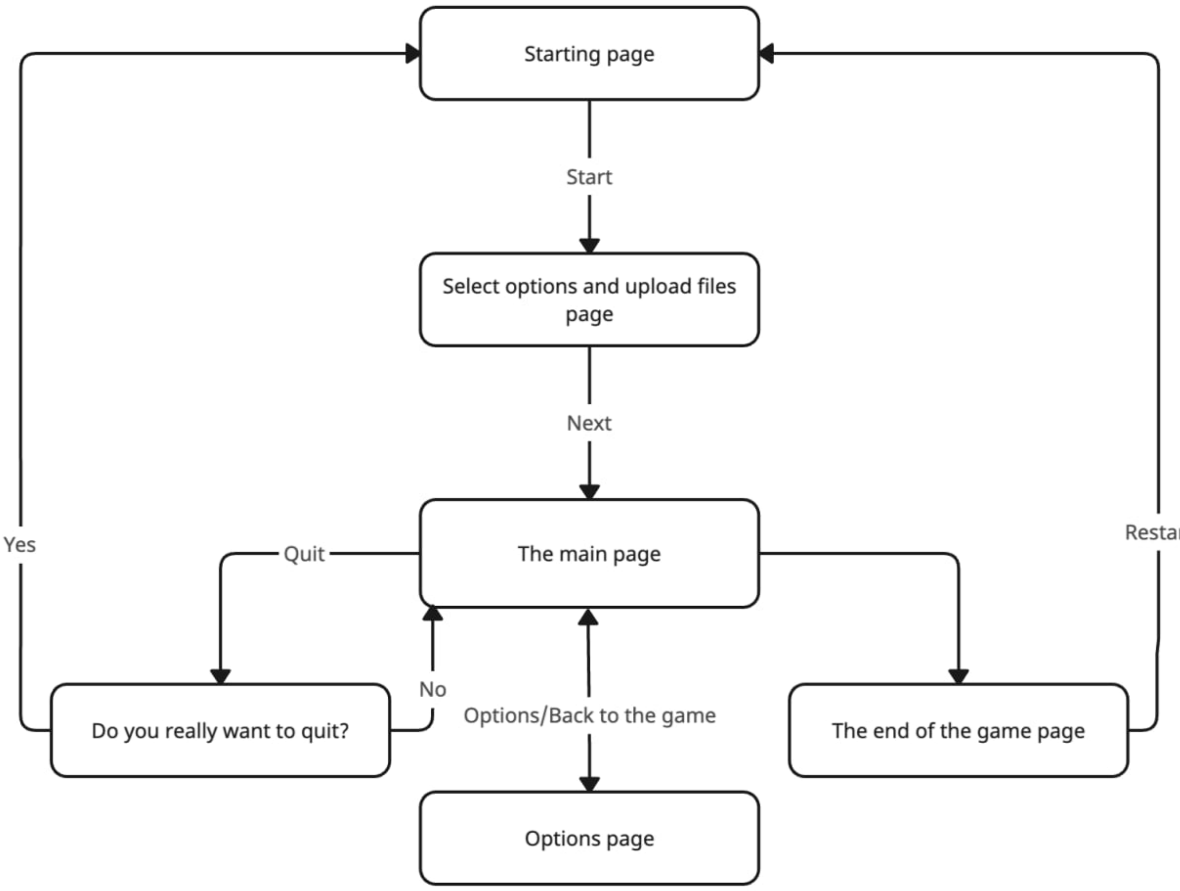
1.1 Bug World App

1.2 The app is designed to simulate the software-defined behavior of bugs located in a custom field, also defined by the user. The behavior of the bug depends on a special assembler-like language program and represents the so-called brain of the bug. The user runs the simulation and observes the beatles.

2. Functional purpose.

- Loading the bug's brain code, i.e. the sequence of bug instructions to execute.
- Loading the field on which the bugs will perform the provided actions.
- Run the experiment twice, swapping the position of the bugs.

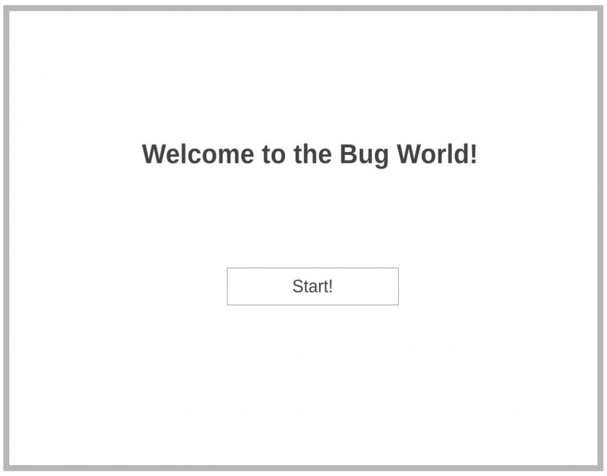
3. Functional requirements

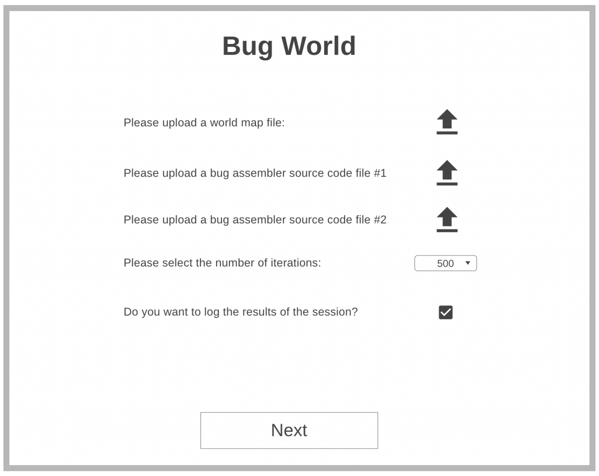


Graph of all transitions between UI states

The start page should look like this and present a start button and welcome to the player (pic. 1)

After clicking on the start button, the user should be taken to a settings page where they can select the world map to load, the code for the first bug team brain, the code for the second bug team brain, the number of iterations and enable/disable game logging (pic. 2)





Please upload a world map file:

Please upload a bug assembler source code file #1

Please upload a bug assembler source code file #2

Please select the number of iterations: 500

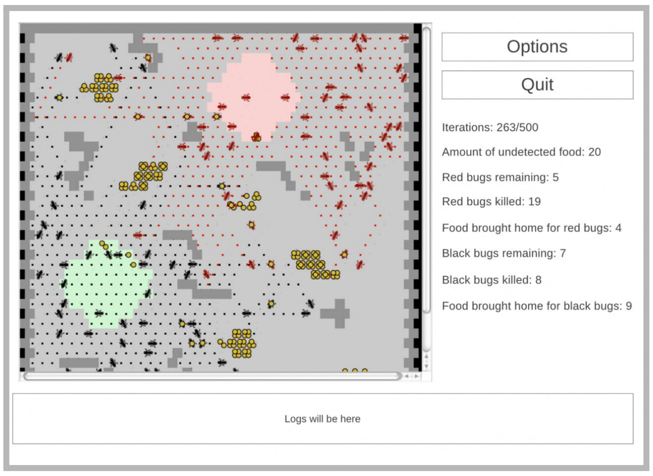
Do you want to log the results of the session? ✓

pic. 1

pic. 2

After setting up, the user should be taken to the home page, which contains the playing field, the game settings and exit buttons, and the game statistics. The game statistics are the number of iterations, the food eaten, the number of bugs left and killed by both teams, and how much food each team has brought to its base. There should be a field on the page, for logging and displaying the current game state (pic. 3)

During the game, the user can change the following settings: the color of each team's bugs, the number of iterations and the real time in which one tick will occur (pic. 4). Bug colors must be distinct and chosen from a predefined set (web standard): aqua, black, blue, fuchsia, gray, green, lime, maroon, navy, olive, purple, red, silver, teal, white, and yellow.



Iterations: 263/500

Amount of undetected food: 20

Red bugs remaining: 5

Red bugs killed: 19

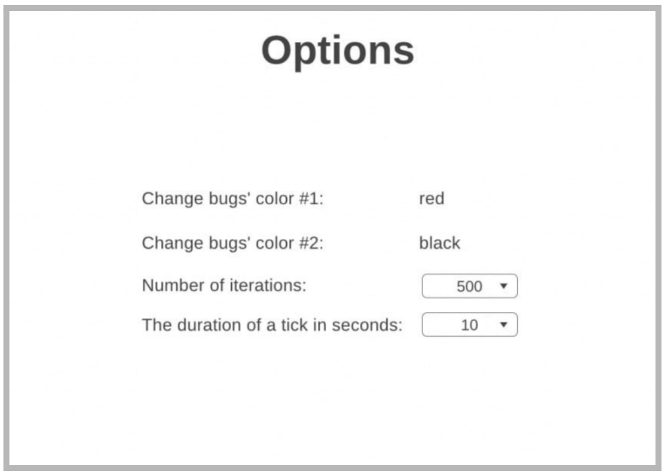
Food brought home for red bugs: 4

Black bugs remaining: 7

Black bugs killed: 8

Food brought home for black bugs: 9

pic. 3



Change bugs' color #1: red

Change bugs' color #2: black

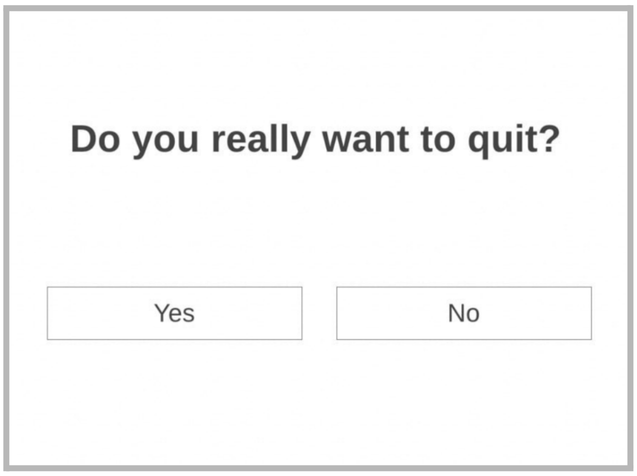
Number of iterations: 500

The duration of a tick in seconds: 10

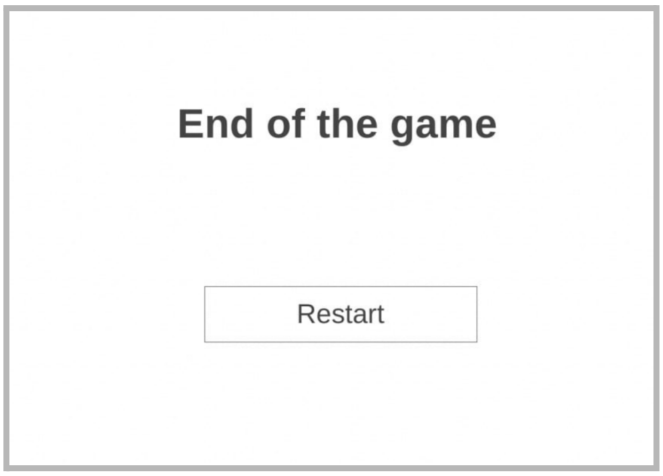
pic. 4

It is possible to leave the game before it ends. The page displays two buttons: yes and no, allowing the user to log out or stay in the game (pic. 5)

After the game itself has ended, the user should be presented with a page with a button that allows them to start the game again (pic. 6)

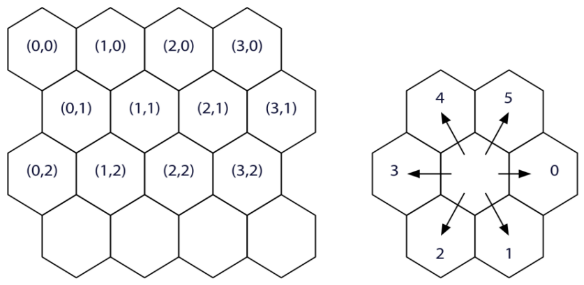


pic. 5



pic. 6

Each cell is a hexagon with six adjacent cells, which are numbered from one to six, starting clockwise from the right (pic. 7)



pic. 7

3.1 Input/Output requirements

3.1.1 User input:

- A world map file
- Two bug assembler source code files, one for the red and one for the black bug; ~~instead of calling the competing bugs red and black, individual names may be provided (eg, by deducing them from the code file names)~~
- Number of iterations the simulation should run
- Optional controls, such as activating log output

3.1.2 Output:

- A visual representation of the world map, updated with every simulation step
- The current iteration counter, and how many iterations are left
- A summary of the current tournament status (amount of undetected food; for red and black bugs: food brought home, bugs killed / remaining; etc.)
- If enabled: log output on screen

4. ~~Non-functional requirements~~ Architecture

4.1 Bug

4.1.1 Bug interaction

- ~~**sensedCell(p,dir)** – determines the target position from position p and absolute heading dir.~~
- ~~**dead(b)** – set bug b state to dead~~
- ~~**position(b)** – return position of bug b~~
- ~~**kill(pos)** – kill bug at position pos and remove it from the corresponding cell~~
- **turnLeft()** -- change bug’s direction one step counterclockwise
- **turnRight()** --- change bug’s direction one step clockwise

4.1.2 Bug requirements

- Bugs can set 6 markers into different cells. Each marker is identified by a number ranging from 0 to 5. Every swarm has its individual set of 6 possible markers.
- A bug can investigate its immediate neighborhood, concretely: its own cell, the cell ahead, left ahead, and right ahead.

4.2 WorldCell

4.2.1 Cell interaction

- **setMarkerAt (~~pos~~,c,i)** – set marker i ~~at position pos~~ for swarm of color c
- **clearMarkerAt(~~pos~~,c,i)** – delete marker i ~~at position pos~~ for swarm of color c

- **isFriendlyMarkerAt(~~pos~~,c,i)** – return true of marker i is set ~~at position pos~~ for color c
- **isEnemyMarkerAt(~~pos~~,c~~,i~~)** – return true if ~~at position pos~~ any marker is set by the adversarial swarm
- **cellMatches(~~pos~~,cond,c)** – returns true if the cell satisfies the conditions, where team with color c is considered “Friendly”
- ~~**adjacent(p,d)** – delivers the cell immediately neighbouring p in direction d.~~

4.2.2 Cell ~~values~~ conditions

- *Friend*: cell is occupied by bug of own color
- *Foe*: cell is occupied by adversarial bug
- *FriendWithFood*: cell is occupied by friendly bug carrying food
- *FoeWithFood*: cell is occupied by adversarial bug carrying food
- *Food*: cell contains food
- *Rock*: cell is an obstacle
- *Marker* i: cell contains own marker with number i
- *FoeMarker*: cell has adversarial marker
- *Home*: cell is part of own nest
- *FoeHome*: cell is part of adversarial nest

4.2.3 Cell requirements

- Before the first iteration the world contains no markers.
- Bugs can differentiate their own markers for their number, for adversarial markers they only can sense their presence, but not their value.

4.3 Bug Brain

4.3.1 Bug Brain interaction

- **sense sensedir s1 s2** – cond check if condition cond is fulfilled in direction sensedir; if yes, go to s1, otherwise s2.
- **mark m s** – set marker m in current cell, then go to s.
- **unmark i s** – delete marker I, then go to s.
- **pickup s1 s2** – take food from current cell, then go to s1; if no food is available or bug already carries food then go to s2.
- **drop s** – put food into current cell and go to s.
- **turn lr s** – turn in direction lr (left or right), then go to s.
- **move s1 s2** – advance by once cell in current direction, then go to s1; if cell ahead is blocked go to s2.
- **flip p s1 s2** – obtain a random number between 0 and p − 1; if zero then go to s1 , otherwise go to s2. direction d s1 s2 if current heading is d then go to s1 , otherwise go to s2.

4.4 World

4.4.1 World interaction

4.4.1.1 World observer

- **size** – returns the world’s dimension in dimensions x and y.
- obstructed – is true if a position is an obstruction and thus cannot carry anything.

- **occupied** – is true if it is holding a bug.
- **getBugAt** – returns the bug of an occupied position. It is a checked run-time error to apply bug at to a cell an unoccupied cell.
- foodAt – returns the amount of food at a position.
- **isFriendlyBaseAt** – is true if a position belongs to bug of a given color.
- **isEnemyBaseAt**– is true if a base of a different color is at a position.
- **isFriendlyMarkerAt** – is true if a position holds markers for a given color of bugs.
- **isEnemyMarkerAt**– is true if a position holds any marker for a color different than the given one.
- **adjacent other bugs** – returns the number of different-colored bugs adjacent to a given position.
- **cellMatches** – if a cell matches a condition cond for a bug of color.
- **getColorStrategy** – returns BugBrain (instructions) for the team of the given color

4.4.1.2 World mutator

- **setFoodAt** – places food (bits) at a position.
- **removeBugAt** – removes a bug from a position; this does not affect the liveness of the bug.
- **setMarkerAt** – sets a marker i for bugs of a certain color.
- **clearMarkerAt** – removes a marker i for bugs of a certain color.
- **setBugAt** – places a bug at a position.

~~4.4.1.3 World logging~~

- ~~**stats(format)** – function returns a record of statistics per bug colony: the number of alive bugs, and the amount of food (e.g. bits) placed on the colony's base. two formats are available: icfp log and sopra log~~

4.3.1.3 World Statistics

World Statistics is a separate class which has access to read World Map (We can give a link to the World in the constructor). World needs to have its own World Statistics object. World Statistics holds all the necessary counters for stats (Current iteration, amount of iterations, killed red/black bags, remain red/black bags, food brought to the red/black bugs home)

methods:

- initialize() – reads World Map and set all the counters (how many bugs, food, etc.)
- update() – it will update counters by analyzing current World Map. This method should be called in the end of every iteration of Simulation
- get*() – getters for all counters

4.4.2 World values (ASCII representation)

- # – obstacle
- . – empty cell (ie, no bug) - empty cell
- - – black swarm nest
- + – empty cell, red swarm nest 1..9 empty cell with number of food units

4.4.3 World requirements

- width of map on first line
- height of map on second line
- A map initially does not contain bugs.
- Before the first iteration every nest cell is populated with a bug of the corresponding color.

4.5 Bug assembler

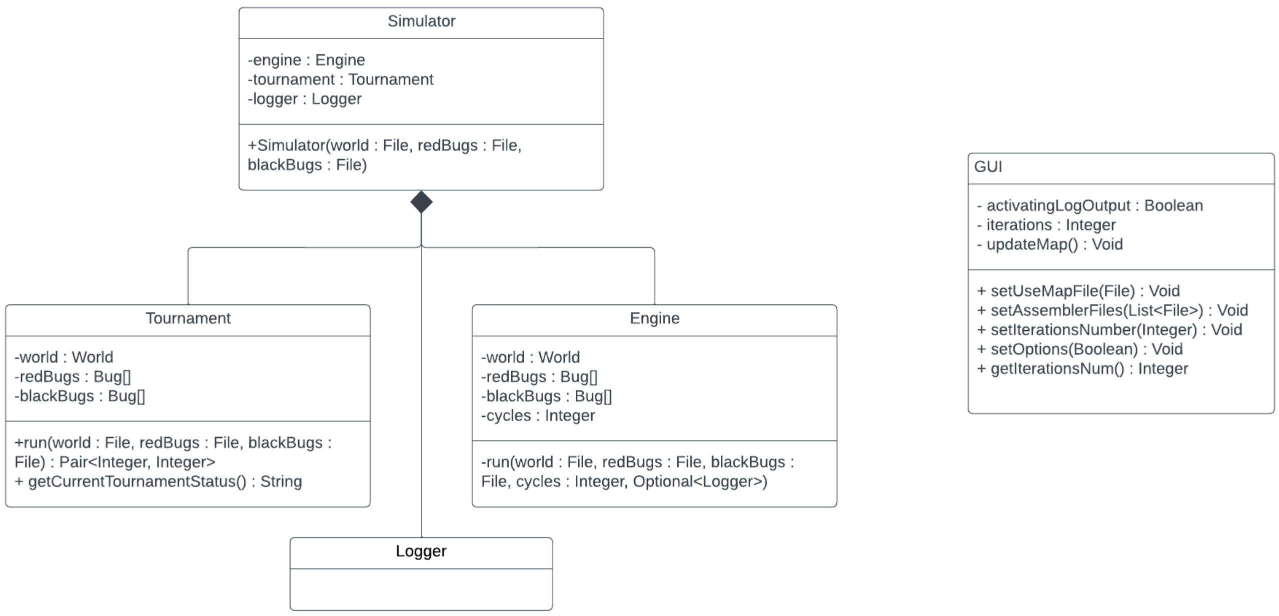
4.5.1 Bug assembler interaction

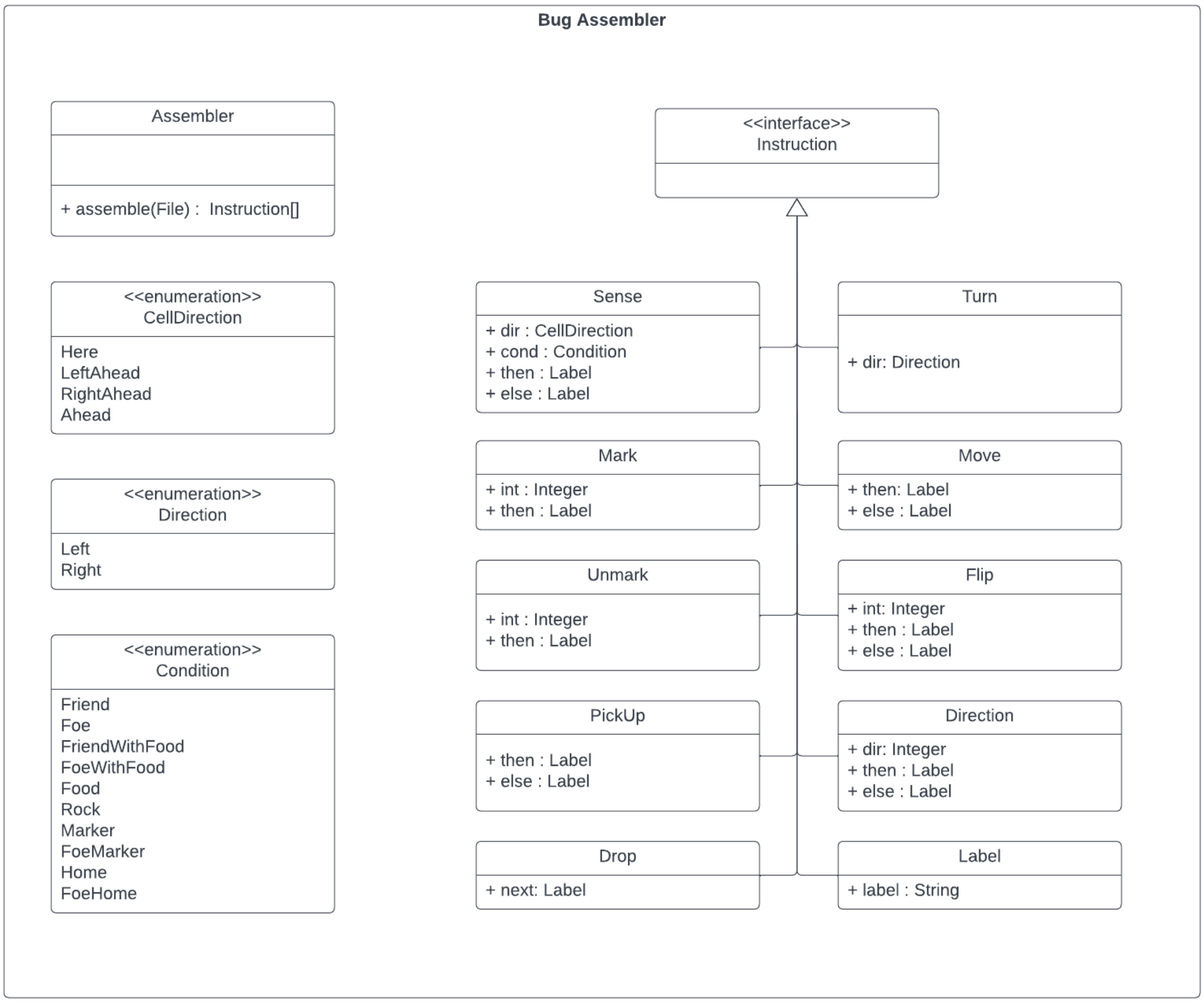
- **assemble()** – function implements the two passes: defining labels and translating the program to machine code

4.5.2 Bug assembler requirements

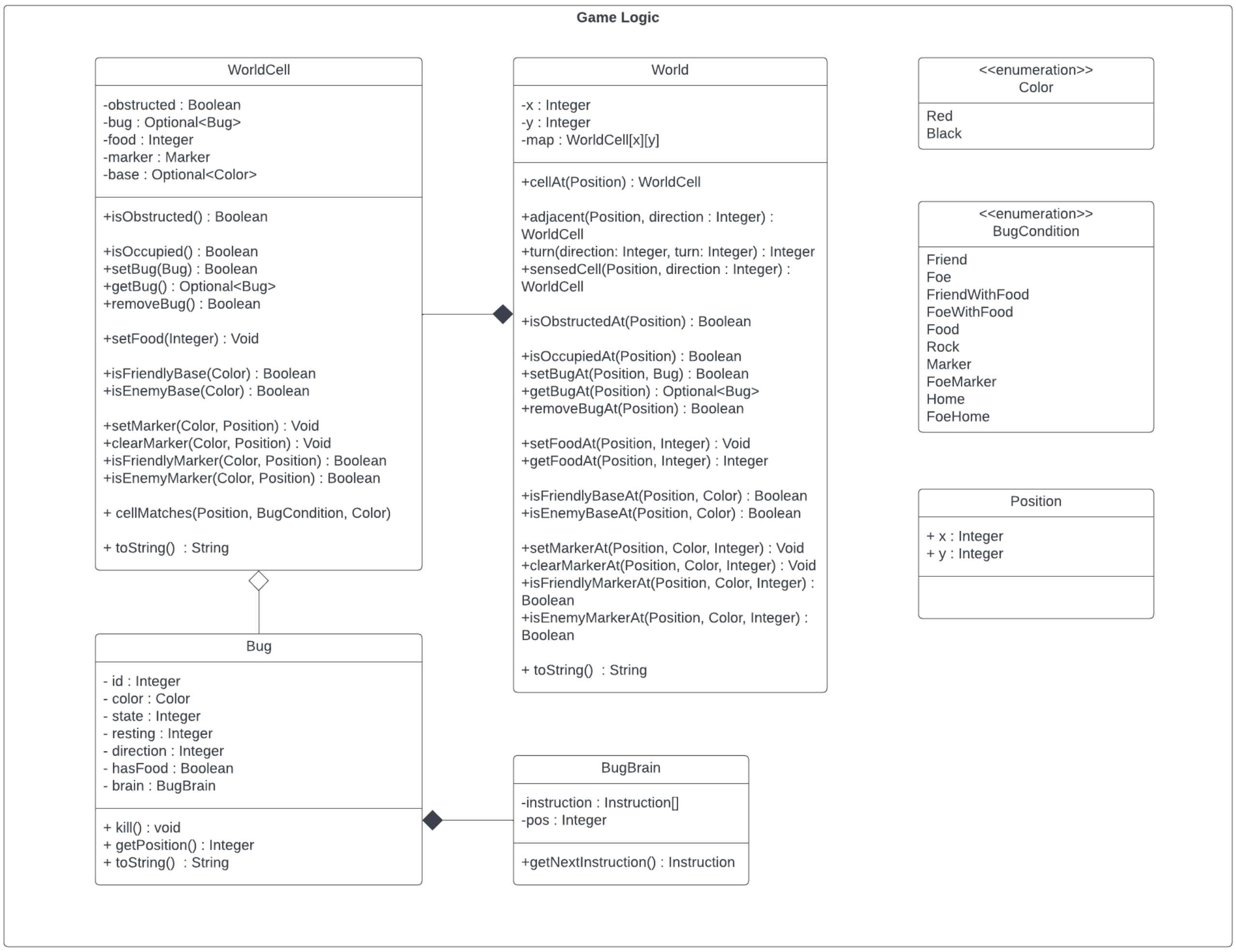
- The first pass assigns addresses to labels (“address resolution”). To this end, it determines the position number of the machine instruction an assembler instruction is translated to in the second pass.
- The second step is the heart of the assembler: the translation of an assembler instruction to a machine instruction.

5. UML diagram

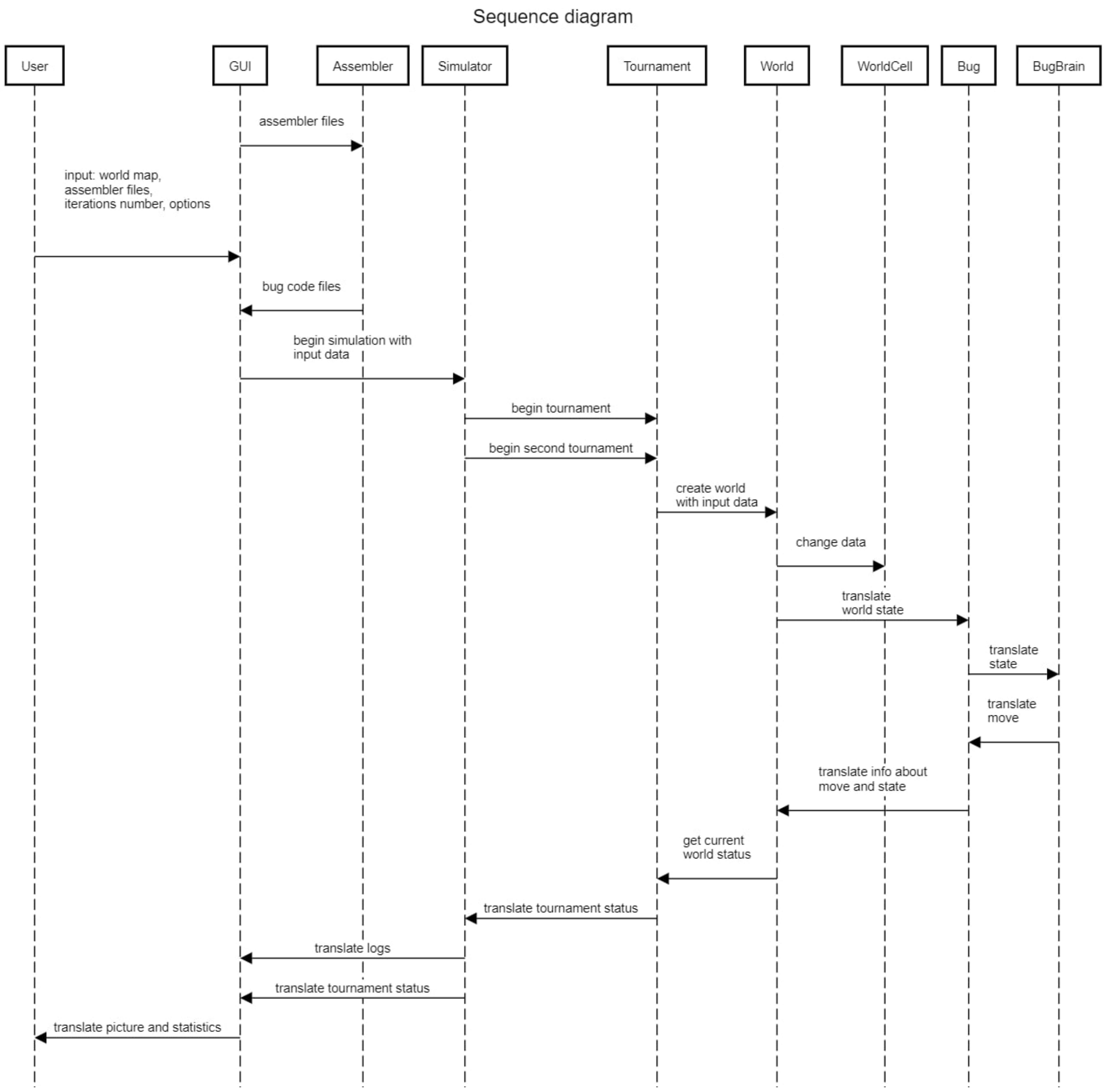

Simulator
-engine : Engine
-tournament : Tournament
-logger : Logger
+Simulator(world : File, redBugs : File, blackBugs : File)
Tournament
-world : World
-redBugs : Bug[]
-blackBugs : Bug[]
+run(world : File, redBugs : File, blackBugs : File) : Pair<Integer, Integer>
+ getCurrentTournamentStatus() : String
Engine
-world : World
-redBugs : Bug[]
-blackBugs : Bug[]
-cycles : Integer
-run(world : File, redBugs : File, blackBugs : File, cycles : Integer, Optional<Logger>)
Logger
GUI
- activatingLogOutput : Boolean
- iterations : Integer
- updateMap() : Void
+ setUseMapFile(File) : Void
+ setAssemblerFiles(List<File>) : Void
+ setIterationsNumber(Integer) : Void
+ setOptions(Boolean) : Void
+ getIterationsNum() : Integer


Bug Assembler
Assembler
+ assemble(File) : Instruction[]
<<enumeration>>
CellDirection
Here
LeftAhead
RightAhead
Ahead
<<enumeration>>
Direction
Left
Right
<<enumeration>>
Condition
Friend
Foe
FriendWithFood
FoeWithFood
Food
Rock
Marker
FoeMarker
Home
FoeHome
<<interface>>
Instruction
Sense
+ dir : CellDirection
+ cond : Condition
+ then : Label
+ else : Label
Turn
+ dir: Direction
Mark
+ int : Integer
+ then : Label
Move
+ then: Label
+ else : Label
Unmark
+ int : Integer
+ then : Label
Flip
+ int: Integer
+ then : Label
+ else : Label
PickUp
+ then : Label
+ else : Label
Direction
+ dir: Integer
+ then : Label
+ else : Label
Drop
+ next: Label
Label
+ label : String

OBJ


Game Logic
WorldCell
-obstructed : Boolean
-bug : Optional<Bug>
-food : Integer
-marker : Marker
-base : Optional<Color>
+isObstructed() : Boolean
+isOccupied() : Boolean
+setBug(Bug) : Boolean
+getBug() : Optional<Bug>
+removeBug() : Boolean
+setFood(Integer) : Void
+isFriendlyBase(Color) : Boolean
+isEnemyBase(Color) : Boolean
+setMarker(Color, Position) : Void
+clearMarker(Color, Position) : Void
+isFriendlyMarker(Color, Position) : Boolean
+isEnemyMarker(Color, Position) : Boolean
+ cellMatches(Position, BugCondition, Color)
+ toString() : String
World
-x : Integer
-y : Integer
-map : WorldCell[x][y]
+cellAt(Position) : WorldCell
+adjacent(Position, direction : Integer) : WorldCell
+turn(direction: Integer, turn: Integer) : Integer
+sensedCell(Position, direction : Integer) : WorldCell
+isObstructedAt(Position) : Boolean
+isOccupiedAt(Position) : Boolean
+setBugAt(Position, Bug) : Boolean
+getBugAt(Position) : Optional<Bug>
+removeBugAt(Position) : Boolean
+setFoodAt(Position, Integer) : Void
+getFoodAt(Position, Integer) : Integer
+isFriendlyBaseAt(Position, Color) : Boolean
+isEnemyBaseAt(Position, Color) : Boolean
+setMarkerAt(Position, Color, Integer) : Void
+clearMarkerAt(Position, Color, Integer) : Void
+isFriendlyMarkerAt(Position, Color, Integer) : Boolean
+isEnemyMarkerAt(Position, Color, Integer) : Boolean
+ toString() : String
<<enumeration>> Color
Red
Black
<<enumeration>> BugCondition
Friend
Foe
FriendWithFood
FoeWithFood
Food
Rock
Marker
FoeMarker
Home
FoeHome
Position
+ x : Integer
+ y : Integer
Bug
- id : Integer
- color : Color
- state : Integer
- resting : Integer
- direction : Integer
- hasFood : Boolean
- brain : BugBrain
+ kill() : void
+ getPosition() : Integer
+ toString() : String
BugBrain
-instruction : Instruction[]
-pos : Integer
+getNextInstruction() : Instruction

6. Sequence diagram



7. Test cases

The program must respond correctly to both correct and incorrect output. To do this, we first classify errors in the input data, then check that the game mechanics work correctly with the correct input data.
Accordingly, we will have three blocks: world errors, bug brain errors, checking basic mechanics

7.1 Wrong world Inputs

First, let's list the errors with the field format and the errors that the program should display on them.

7.1.1 Error: The field does not correspond to the indicated dimensions

```
10
11
                         #  #  #  #  #  #  #  #  #  #
                         #  9  9  .  .  .  .  3  3  #
                         #  9  #  .  .  .  .  .  .  #
                         #  .  #  .  .  .  .  .  .  #
                         #  .  .  5  .  .  .  .  .  #
                         #  +  +  +  +  +  5  .  .  #
                         #  +  +  +  +  +  +  #  .  #
                         #  +  +  +  +  +  .  #  9  #
                         #  3  3  .  .  .  .  9  9  #
                         #  #  #  #  #  #  #  #  #  #
```

7.1.2 Error: One of the bug swarms is missing

```
10
10

                         #  #  #  #  #  #  #  #  #  #
                         #  9  9  .  .  .  .  3  3  #
                         #  9  #  .  .  .  .  .  .  #
                         #  .  #  .  .  .  .  .  .  #
                         #  .  .  5  .  .  .  .  .  #
                         #  +  +  +  +  +  5  .  .  #
                         #  +  +  +  +  +  +  #  .  #
                         #  +  +  +  +  +  .  #  9  #
                         #  3  3  .  .  .  .  9  9  #
                         #  #  #  #  #  #  #  #  #  #
```

7.1.3 Error: There is no outer border

10
10

```
# # # # # # # # # #
# 9 9 . . . . 3 3 #
# 9 # . - - - - - #
# . # - - - - - - #
# . . 5 - - - - - #
# + + + + + + . . #
# + + + + + + # . #
# + + + + + . # 9 #
# 3 3 . . . . 9 9 .
# # # # # # # # # #
```

7.1.4 Error: Value out of legal

10
10

```
# # # # # # # # # #
# a 9 . . . . 3 3 #
# 9 # . - - - - - #
# . # - - - - - - #
# . . 10 - - - - - #
# + + + + + 5 . . #
# + + + + + + # . #
# + + + + + . # 9 #
# 3 3 . . . . 0 9 #
# # # # # # # # # #
```

7.1.5 Error: Swarm have to be linked

```
10
10
# # # # # # # # # #
# 9 9 . . . . 3 3 #
# 9 # . - - - - - #
# . # - - - - - - #
# . . 5 - - - - - #
# + + + . + 5 . . #
# + + + . + + # . #
# + + + . + . # 9 #
# 3 3 . . . . 9 9 #
# # # # # # # # # #
```

7.2 Wrong bug input

In a similar format, errors that a user might make in describing a bug's brain

7.2.1 Error: link to a non-existent line

```
sense ahead 1 3 food ; [ 0]
move 2 0 ; [ 1]
pickup 8 0 ; [ 2]
flip 3 4 5 ; [ 3]
turn left 0 ; [ 4]
flip 2 6 1223674 ; [ 5]
turn right 0 ; [ 6]
move 0 3 ; [ 7]
sense ahead 9 11 home ; [ 8]
move 10 8 ; [ 9]
drop 0 ; [10]
flip 3 12 13 ; [11]
turn left 8 ; [12]
flip 2 14 15 ; [13]
turn right 8 ; [14]
move 8 11 ; [15]
```

7.2.2 Error: typos/non-existent tokens

```
sense unrecognizable_word 1 3 food ; [ 0]
```

```
 move 2 0 ; [ 1]
 pickup 8 0 ; [ 2]
 flip 3 4 5 ; [ 3]
 turn left 0 ; [ 4]
 flip 2 6 7 ; [ 5]
 turn right 0 ; [ 6]
 move 0 3 ; [ 7]
 sense ahead 9 11 home ; [ 8]
 move 10 8 ; [ 9]
 drop 0 ; [10]
 flip 3 12 13 ; [11]
 turn left 8 ; [12]
 flip 2 14 15 ; [13]
 turn right 8 ; [14]
 move 8 11 ; [15]
```

7.2.3 Error: incorrect use of the command / missing token

```
sense ahead 1 3 food ; [ 0]
move 2 ; [ 1] absent value
pickup 8 0 ; [ 2]
flip 3 4 5 ; [ 3]
turn left 0 ; [ 4]
flip 2 6 7 ; [ 5]
turn right 0 ; [ 6]
move 0 3 ; [ 7]
sense ahead 9 11 home ; [ 8]
move 10 8 ; [ 9]
drop 0 ; [10]
flip 3 12 13 ; [11]
turn left 8 ; [12]
flip 2 14 15 ; [13]
turn right 8 ; [14]
move 8 11 ; [15]
```

7.2.4 Error: lack of commands

```
; there is nothing else here, just this comment
```

7.3 Basic mechanics.

When both the field is correct and the bugs crawl normally, we will check the basic mechanics. Simple examples to test that a particular core game mechanic in isolation works

7.3.1 Food

```
5
5
# # # # #
# # # - #
# # # # #
# + . 9 #
# # # # #
```

```
move 1 0; [0]
sense ahead 2 1 food ; [ 1]
move 3 0 ; [ 2]
pickup 4 0 ; [ 3]
turn left 5 ; [ 4]
turn left 6 ; [ 5]
turn left 7 ; [6]
move 8 0 ; [7]
move 9 0 ; [8]
drop 10 ; [9]
turn right 11 ; [10]
turn right 12 ; [11]
turn right 0; [12]
```

This state machine tells the bug to go back and forth and carry food.

as a result, there will be a field in which the beetle transferred all the food to its base

7.3.2 Rock

```
4
4
# # # #
# # - #
# + # #
# # # #
```

```
move 1 0; [0]
turn left 0; [1]
```

A simple check that the barriers are working. Bugs should stay in their place and only spin

7.3.3 Marks

10
5

```
# # # # # # # # # #
# # # # # # # # - #
# # # # # # # # # #
# + . . . . . . . #
# # # # # # # # # #
```

```
move 1 0; [0]
mark 3 0; [1]
```

The bug must go along the corridor and put marks on each cell

7.3.4 Full game test

10
10

```
# # # # # # # # # #
# 9 9 . . . . 3 3 #
# 9 # . - - - - - #
# . # - - - - - - #
# . . 5 - - - - - #
# + + + + + 5 . . #
# + + + + + + # . #
# + + + + + . # 9 #
# 3 3 . . . . 9 9 #
# # # # # # # # # #
```

```
sense ahead 1 3 food ; [ 0]
move 2 0 ; [ 1]
pickup 8 0 ; [ 2]
flip 3 4 5 ; [ 3]
turn left 0 ; [ 4]
flip 2 6 7 ; [ 5]
turn right 0 ; [ 6]
move 0 3 ; [ 7]
sense ahead 9 11 home ; [ 8]
move 10 8 ; [ 9]
drop 0 ; [ 10]
flip 3 12 13 ; [ 11]
turn left 8 ; [ 12]
flip 2 14 15 ; [ 13]
turn right 8 ; [ 14]
move 8 11 ; [ 15]
```

A complete game with a simple bug algorithm to check step by step that everything is correct