aML

Sriramkumar Balasubramanian
Evan Drewry
Timothy Giel
Nikhil Helferty

Overview

- aML – “a-Mazing Language”
- Can be used to solve mazes by feeding instructions to a bot which is located at the entrance to the maze
- The maze can either be defined by the user in the form of text files or can be randomly generated by the standard library functions

Overview (cont.)

- The language serves as an instruction set to the bot, hence the movement of the bot determines accessing of various data
- AML is designed to not only make the process of solving mazes easier for a programmer, but also to introduce programming to the common man through mazes

AML Tutorial

A brief introduction to syntax

AML Tutorial

- Java/C-like syntax (not exact) enabling you to move a bot around a maze
- Use functions, data types for more complex behavior than just a sequence of moves
- AML provides a visualization of a bot with your program navigating the maze
- Maze provided in .txt file or randomized

AML Tutorial

- Have a limited set of available datatypes
  - Integer
  - Boolean
  - Cell
  - List<datatype> (FIFO)
- Functions can either return a variable type (x():Integer { }) or be *void*
- Can take parameters as well
- The *main* function must be void, parameterless

AML Tutorial

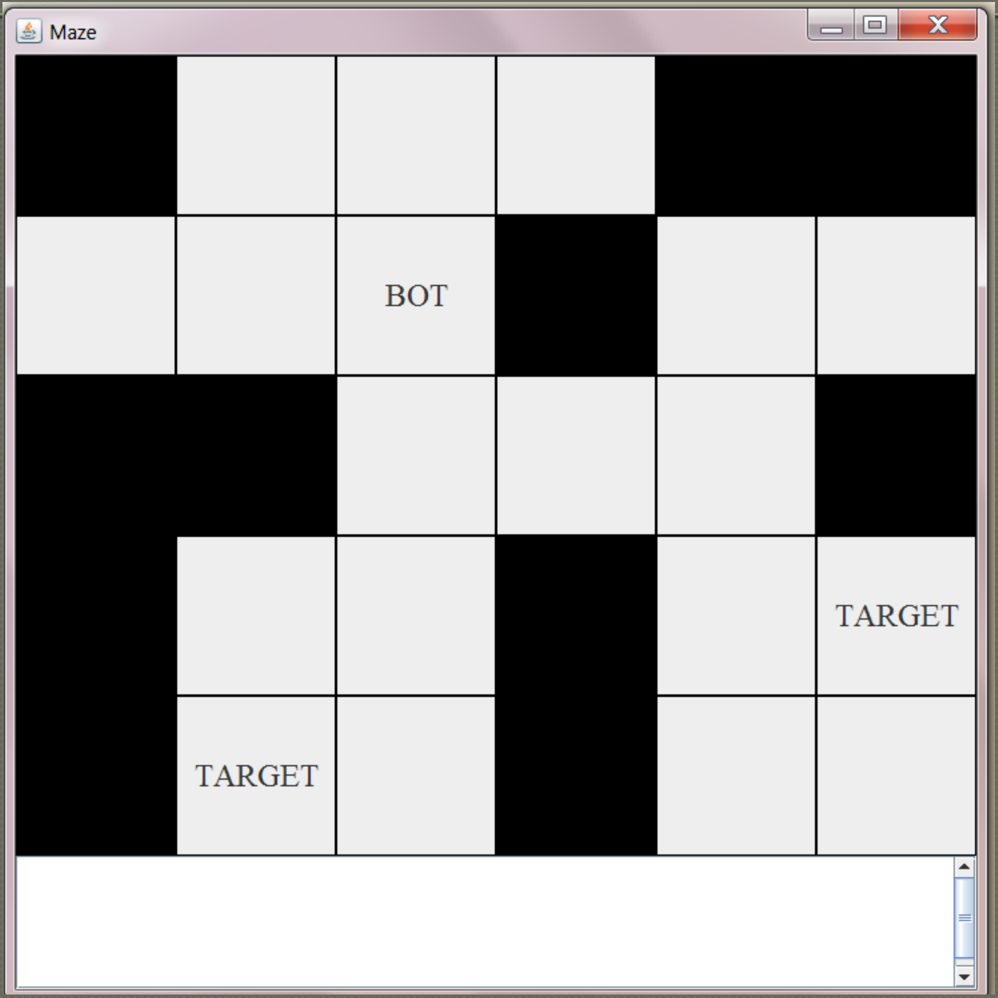
- Maze text format:

```
5 6

0 1 1 1 0 0
1 1 2 0 1 1
0 0 1 1 1 0
0 1 1 0 1 3
0 3 1 0 1 1
```

- First two numbers are # rows and # columns
- Then an integer follows for every cell in row x columns maze
- 0's are "holes"
- 1's are "walkable" cells
- 2 is the start point (only one)
- 3's are targets (multiple possible)

AML Tutorial

```
5 6

0 1 1 1 0 0
1 1 2 0 1 1
0 0 1 1 1 0
0 1 1 0 1 3
0 3 1 0 1 1
```



AML Tutorial

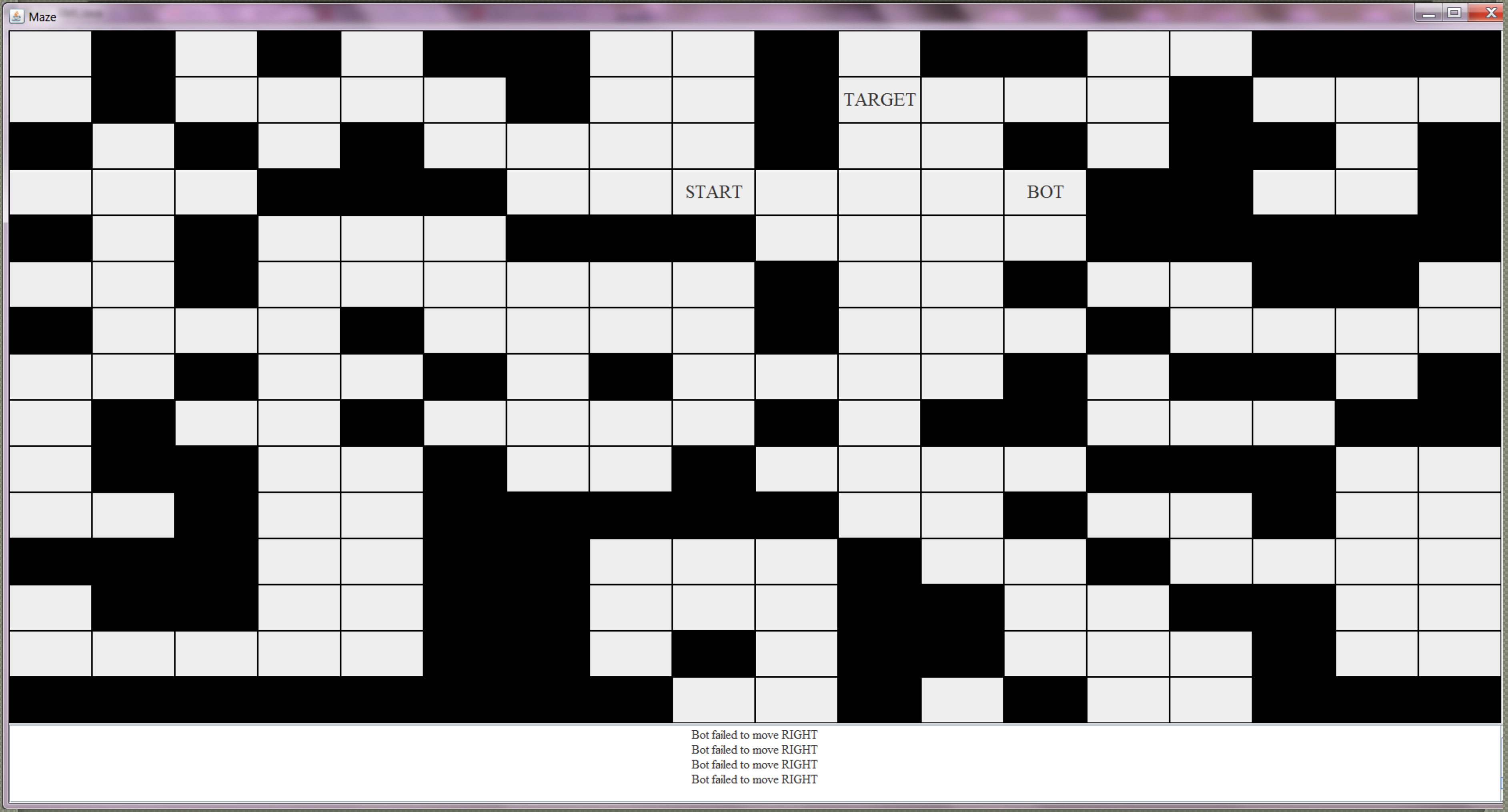
- A very dumb bot:

```
#load-random

// function that is run by program initially
main():void {
		goRight();
}

function goRight():void {
		cell c := (CPos); // variables at start
		move_R(); // moves the bot to the
right

		if (NOT isTarget(c)) {
				goRight();
		};
}
```

How to compile

- (Run “make” to construct AML)
- Run aml on .aml source (for example, aml -c example.aml)
- Run the newly created java code: java example

AML Tutorial



```
Bot failed to move RIGHT
Bot failed to move RIGHT
Bot failed to move RIGHT
Bot failed to move RIGHT
```

Mazes apart … GCD

```
#load-random

main():void{
        integer x := gcd(7,49);
        print(x);
        exit();
}

function gcd(integer n, integer m):integer{
        if(n = m){
                return n;
        }
        else{
                if (n > m) {
                        return gcd(n - m, m);
                }
                else{
                        return gcd(m - n,n);
                }
        }
}
```

Some points to note

- AML will not stop your bot from looping aimlessly into oblivion
  - Could have prevented this possibility in previous program by, for example, limiting the number of attempts with an Integer
- Can design much more complex functions using Lists, recursion, bot's "memory"
- Use the revert() function to backtrack

AML Implementation

Creating the system

Architectural Design

1. Lexical Analyzer
2. Parser
3. Semantic Analysis
4. Translator
5. Top-level

Some Implementation Specifics

- assignment – type consistency
- function calls – two pass run
- Unique main and function definitions checking
- Checking for return statements inside "if's"
- Functions – actual and formal parameters
- **Validity Checking:** Program -> Function -> Statement list -> Statement -> Expression

Lessons Learned

Do’s and Don’ts for the future

Lessons Learned

- Start early
- Split up work s.t. team members aren’t blocking each others progress
- Keep repository updated, use incremental development style
- Don’t plan for “a lot” of features prematurely

Lessons Learned

- Unit testing
- Figure out what tools exist and use them!
  - OCAMLRUNPARAM='p’
  - ocamldep for makefiles
- Don’t assume anything about your teammates; figure out their strengths and split up the work accordingly