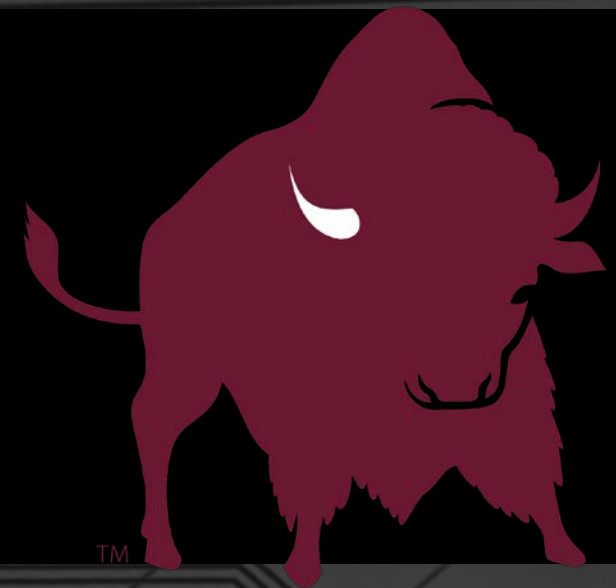


# Thunder the Robot

Panhandle STEM Conference July 23-24, 2018



Presented by H. Paul Haiduk  
hhaiduk@wtamu.edu

Computer Science Program Coordinator  
School of Engineering, Computer Science  
and Mathematics  
West Texas A&M University  
Canyon, TX  
(806) 651-2450



So . . . if you wish to work along with this presentation, proceed to

<https://github.com/HHaiduk/thunder>

- Windows users: Use https download to download **thunder-setup.exe**. Click on that file to start the install process. When the dialog prompts you, simply click Install to accept the default installation location.
- Mac users: Use https download to download the file **ThunderMAC.app.zip**. Extract contents of the zip file somewhere and then copy **Thunder.app** to your Applications directory.
- Linux users: Use https download to download the file **ThunderLINUXsetup**. Ensure that the file is executable. Execute it to complete the installation.

And . . . If you want the examples

Use `https` download to download the file

**`thunder-master.zip`**

You should extract contents of the zip file to some location that you will know about. The extraction process will create the directory/folder **`thunder-master`**.

# Attribution to Richard Pattis

- Karel the Robot *circa* 1981 for the seminal book ***Karel the Robot***.
- The word robot comes from the Czech word *robota*, meaning drudgery or slave-like labor. It was first used to describe fabricated workers in a fictional 1920's play by Czech author Karel Capek called *Rossum's Universal Robots*. According to Pattis, the name Karel the Robot was derived from that early fictional concept and author.
- The work as presented today is motivated by the desire to keep alive one of the most profound concepts in computing and one from which literally hundreds of H. Paul Haiduk's students have begun their journey into that facet of computational thinking called algorithmic thinking.
- Original version of the software written in Pascal and was designed to lead students into the world of Pascal programming. Subsequent versions have been focused on C++ and also Java.

# Thunder the Robot

- Named so to recognize the mascot for WTAMU . . . a buffalo named Thunder.
- Designed to be a “gentle” introduction to the art of algorithmic thinking and programming
- Designed to be a gentle introduction to text-based programming with syntax very similar to Python syntax
- Written in Java with a “look and feel” designed to be essentially the same in Linux, Mac and Windows – any platform that supports the Java Virtual Machine
- Subsequent work planned to render software completely platform independent by use of JavaScript and HTML 5.
- Next slide is the GUI for Thunder . . .

Waiting for world input...

Control Panel

Start

Step

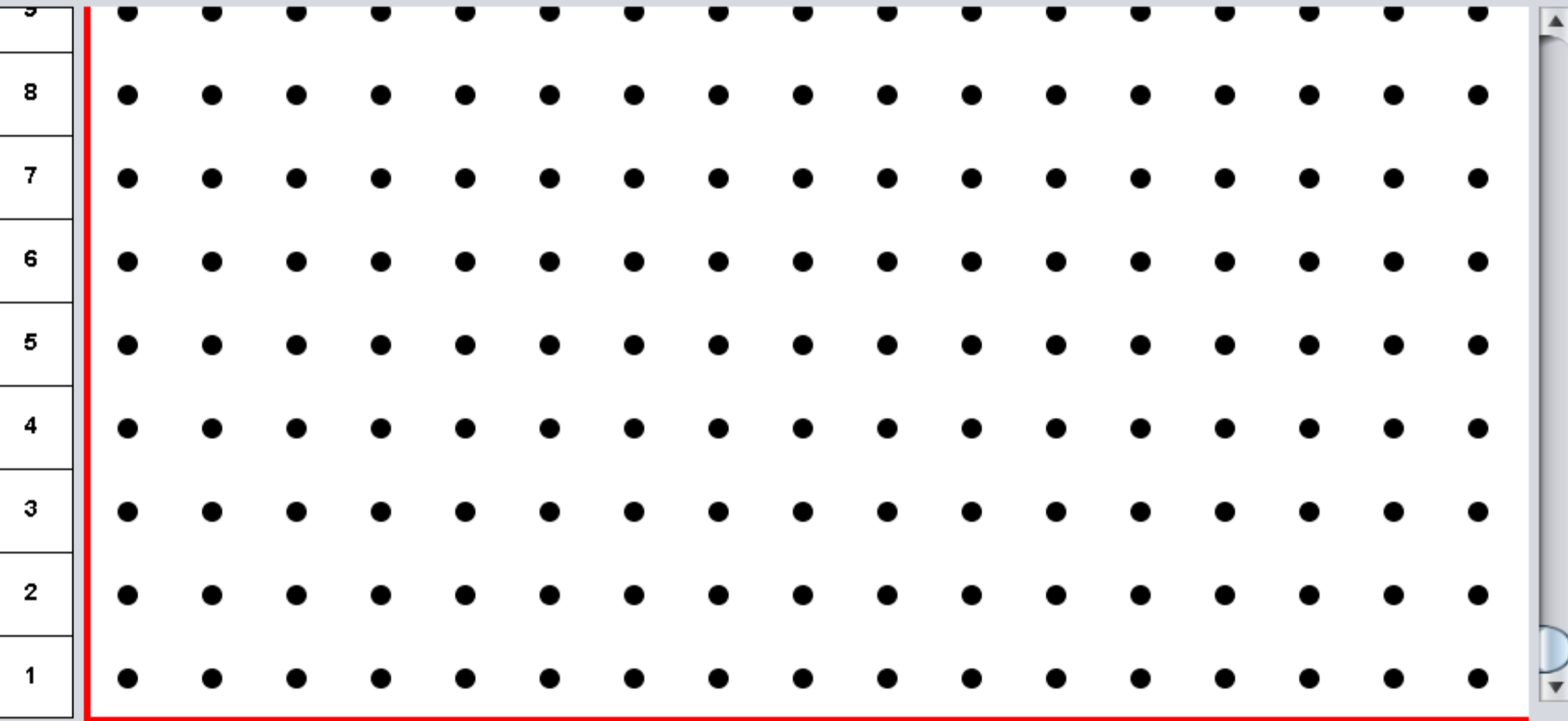
Stop

Reset

Speed: 100%

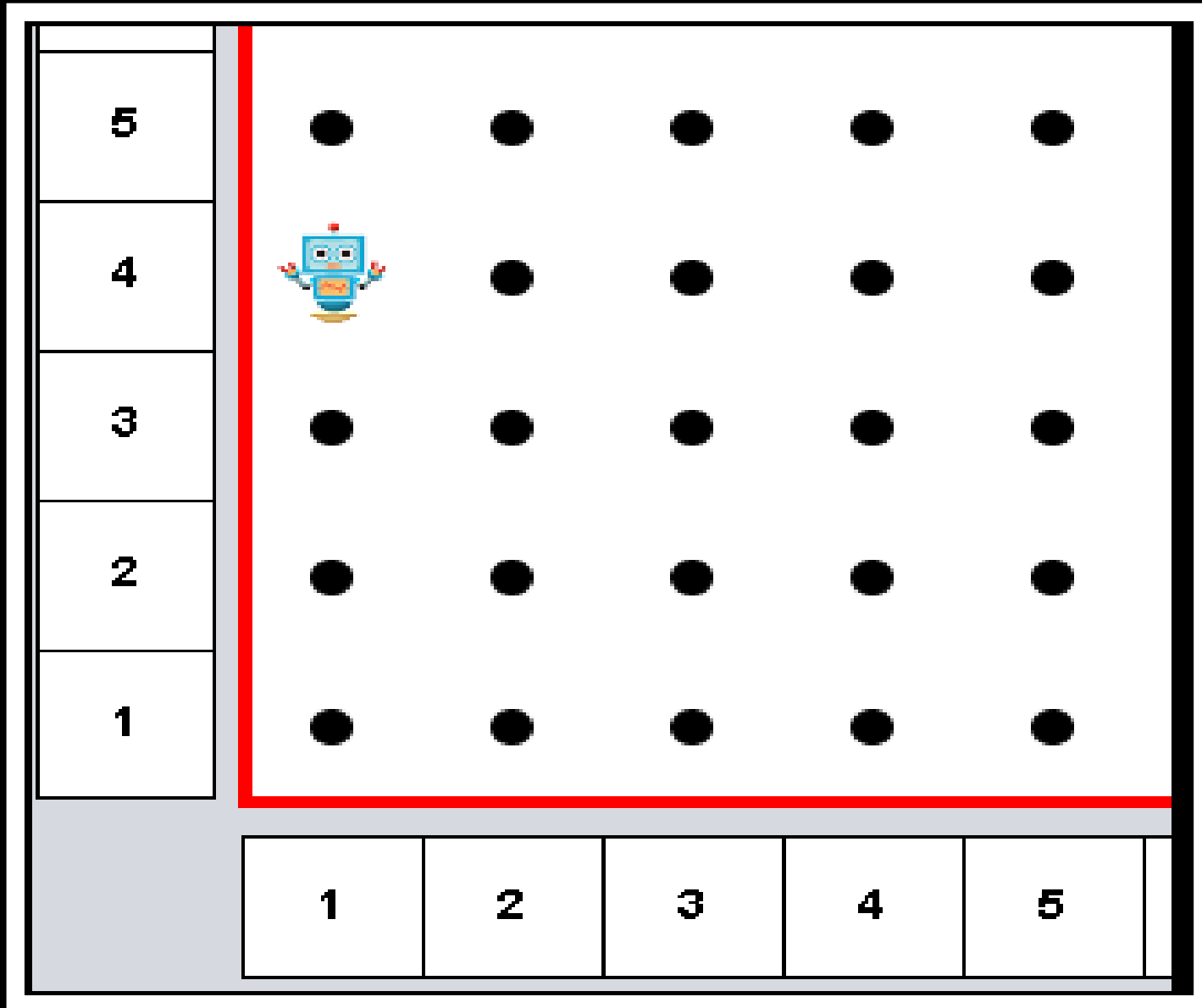
Load World

Split



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----

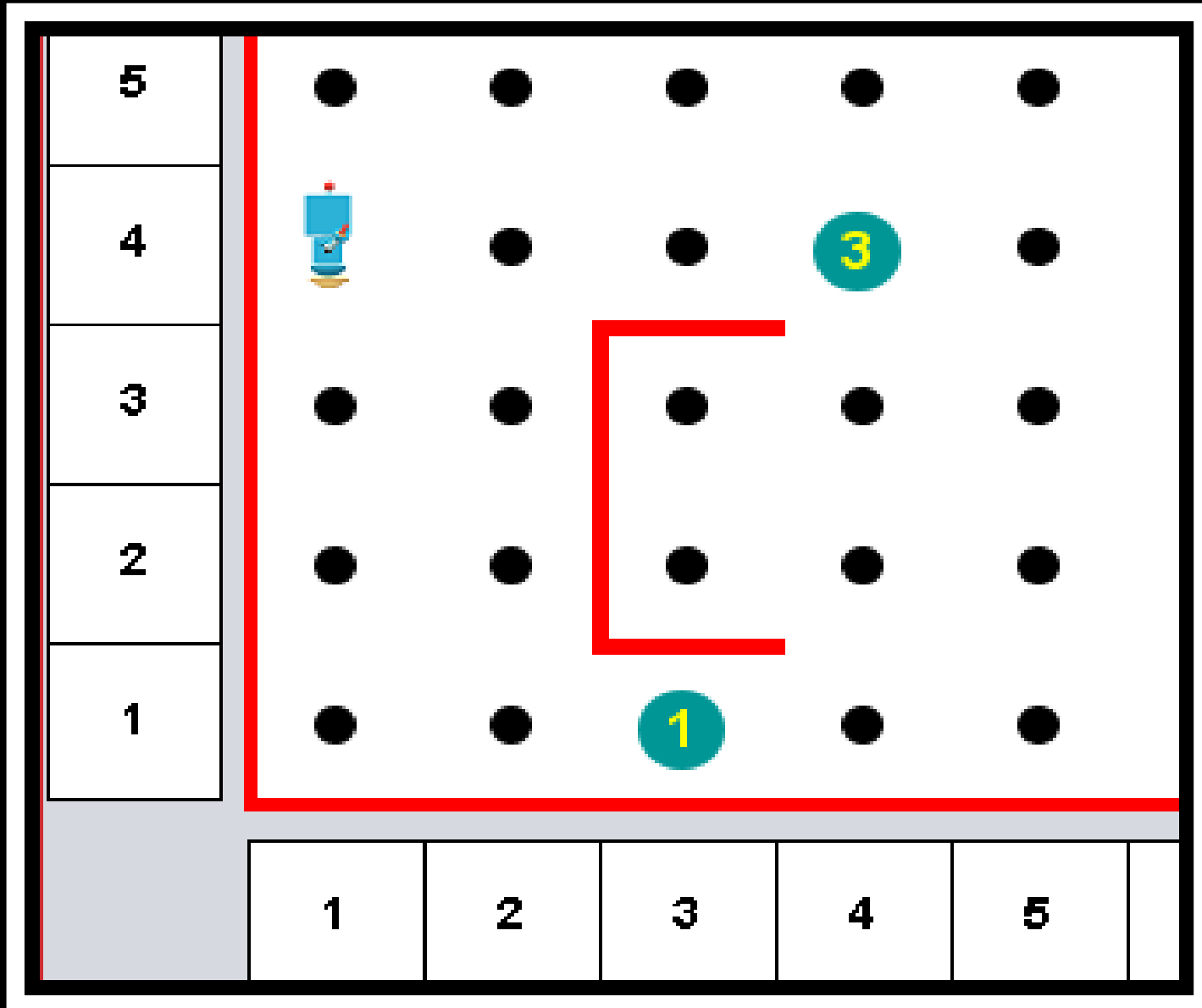
1



## Thunder's World from the “origin”

- Dots represent intersection of streets and avenues
- The robot is located at first avenue and fourth street
- Think of north/south orientation as the Avenues while the streets are east/west
- The robot is facing south.
- This world is finite with 1,000 avenues and 1,000 streets
- Think  $x$  = avenue and  $y$  = street





## Thunder's World (more)

- World is bordered with impenetrable walls
- We may build more walls in the world to pose challenges to the robot's travel – these walls are placed between streets and/or avenues. Also note that these interior walls are also impenetrable
- We may place beepers (little devices that emit a sound) at intersections. Note that the robot must be immediately over the beeper to “hear” its sound.

- **move** -- exactly one block in direction Thunder is facing -- this can cause Thunder to explode if it strikes a wall. Note that Thunder can only move north, east, south or west.
- **turnleft** -- rotate left exactly  $90^\circ$  -- Thunder can always accomplish this no matter where it is located
- **pickbeeper** -- pick up one beeper and place it in a “virtual” beeper bag – this may cause a runtime failure if there is no beeper to pick up
- **putbeeper** – retrieve a beeper from the “virtual” beeper bag and place it at the current intersection – this may cause a runtime failure if there is no beeper in the beeper bag
- **turnoff** – this instruction may be executed at any time and terminates the program in which it exists – the instruction must terminate any given Thunder program

## Thunder's builtin capabilities

Consider these the primitives of the language supporting the robot

- front\_is\_clear
- front\_is\_blocked
- left\_is\_clear
- left\_is\_blocked
- right\_is\_clear
- right\_is\_blocked

## Thunder's sensors

Wall detection tests

- next\_to\_a\_beeper
- not\_next\_to\_a\_beeper
- any\_beeper\_in\_beeper\_bag
- no\_beeper\_in\_beeper\_bag

## Thunder's sensors

Beeper detection tests

- facing\_north
- not\_facing\_north
- facing\_south
- not\_facing\_south
- facing\_east
- not\_facing\_east
- facing\_west
- not\_facing\_west

## Thunder's sensors

Direction detection tests



The World

Language Reference

World Editor Reference

\*Code editor

\*World editor

Waiting for world input...

Control Panel

Start

Step

Stop

Reset

Speed:

100%



Load World

Split

7

6

5

4

3

2

1

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

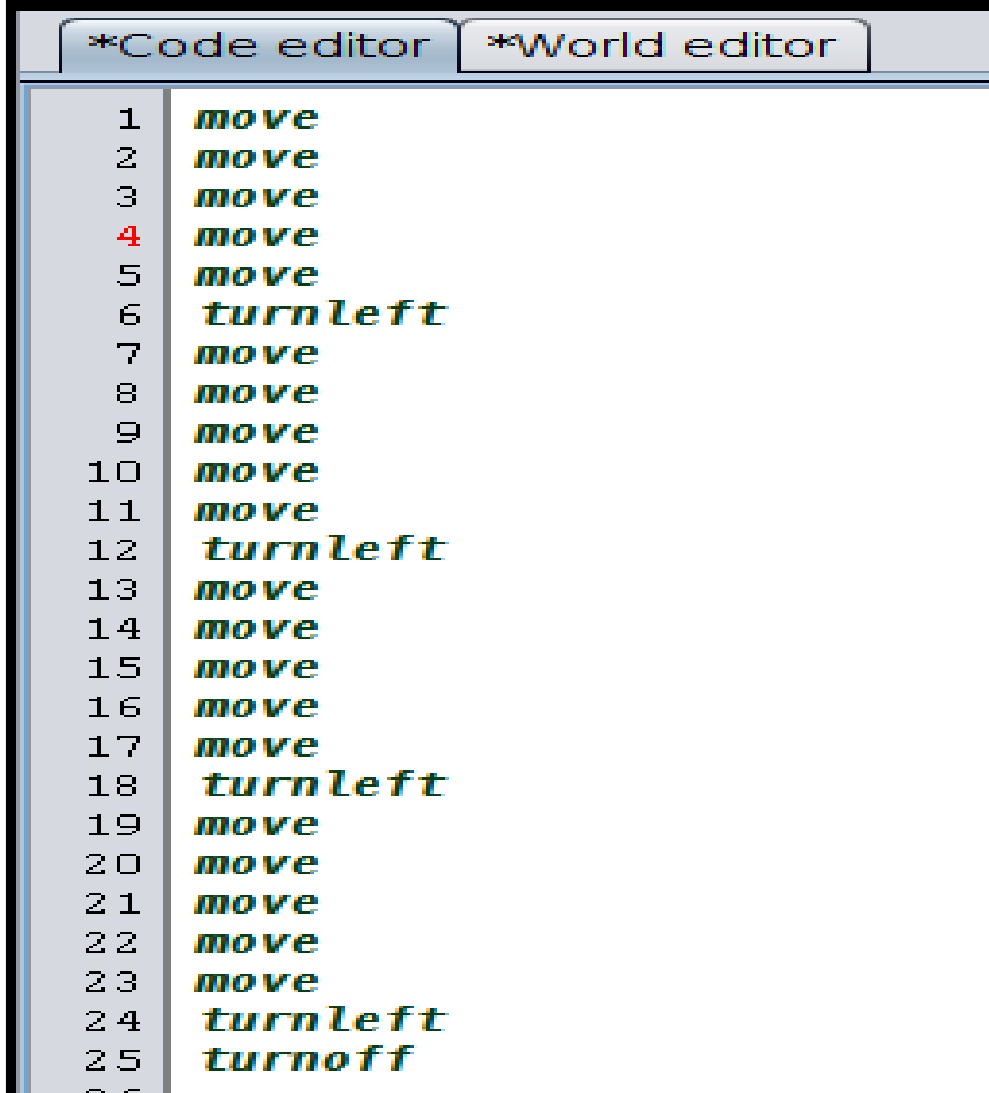
1

# Simple Thunder world

- First we must create the world in which Thunder will operate.
- Click on the World Editor tab in the application and enter/edit the text to appear as:

```
ROBOT 1 1 E 0
```

- This directs the application to “draw” Thunder at 1<sup>st</sup> Avenue and 1<sup>st</sup> Street, facing East, with 0 beepers in the beeper bag.
- Next, click the Load World button in the Control Panel to cause that initial scenario to be displayed. **Please note positioning Thunder in the world is the only manual coding required for the world builder. Creation of walls and beepers can all be accomplished with a mouse in real time.**



The screenshot shows a code editor window with two tabs: '\*Code editor' and '\*World editor'. The '\*Code editor' tab is active, displaying a list of 25 numbered lines of code. The code consists of 'move' and 'turnleft' commands, with the final command being 'turnoff'. Line 4 is highlighted in red. The code is as follows:

```
1  move
2  move
3  move
4  move
5  move
6  turnleft
7  move
8  move
9  move
10 move
11 move
12 turnleft
13 move
14 move
15 move
16 move
17 move
18 turnleft
19 move
20 move
21 move
22 move
23 move
24 turnleft
25 turnoff
```

# Simple Thunder program

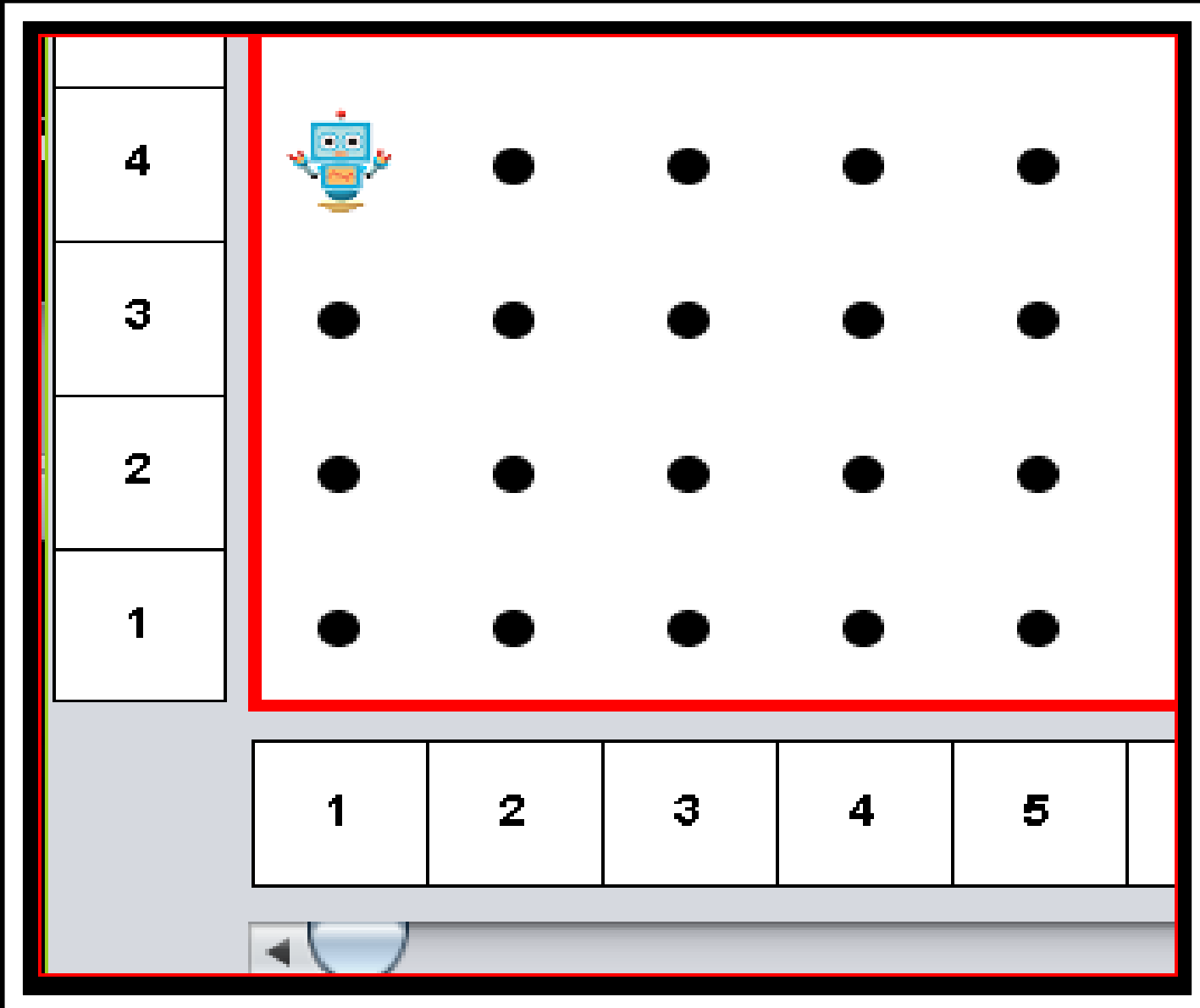
Traverse a path that is five by five and turn off.

To avoid typing this in, first Split the editor pane away from the application pane and click File/Open Source File and navigate to the directory/folder `thunder-master` and load `program01.thunder`

One should also File/Open World File and load `program01.world`.

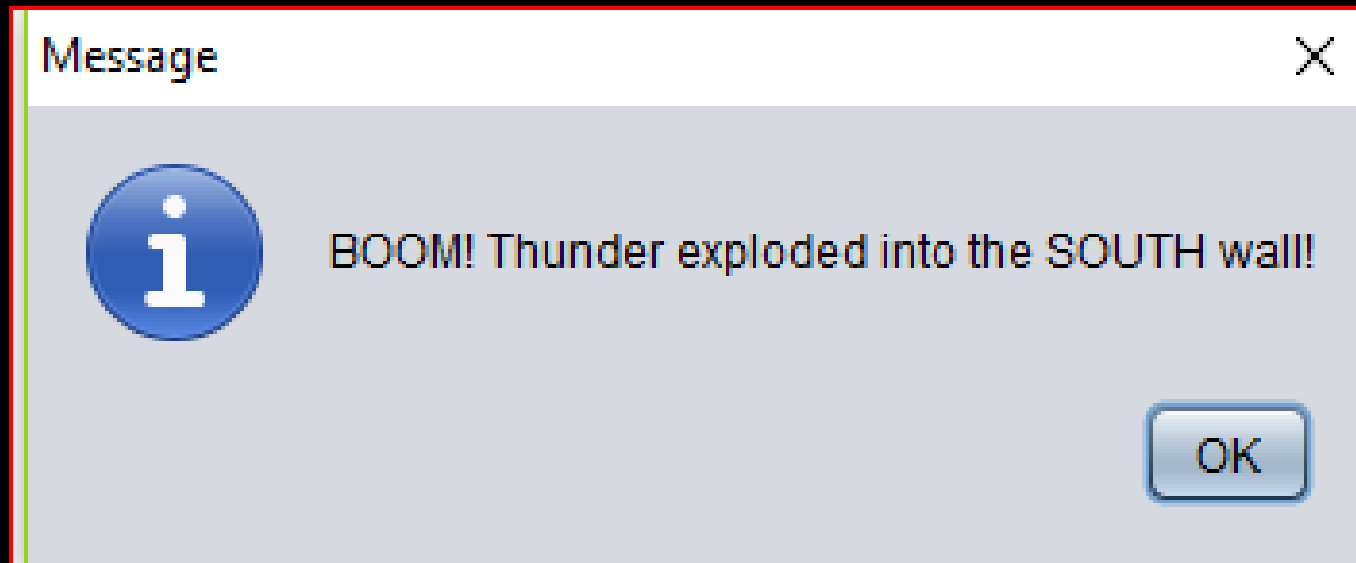
One can then launch the program by clicking Start in the Control Panel or single-step through the program with Step.





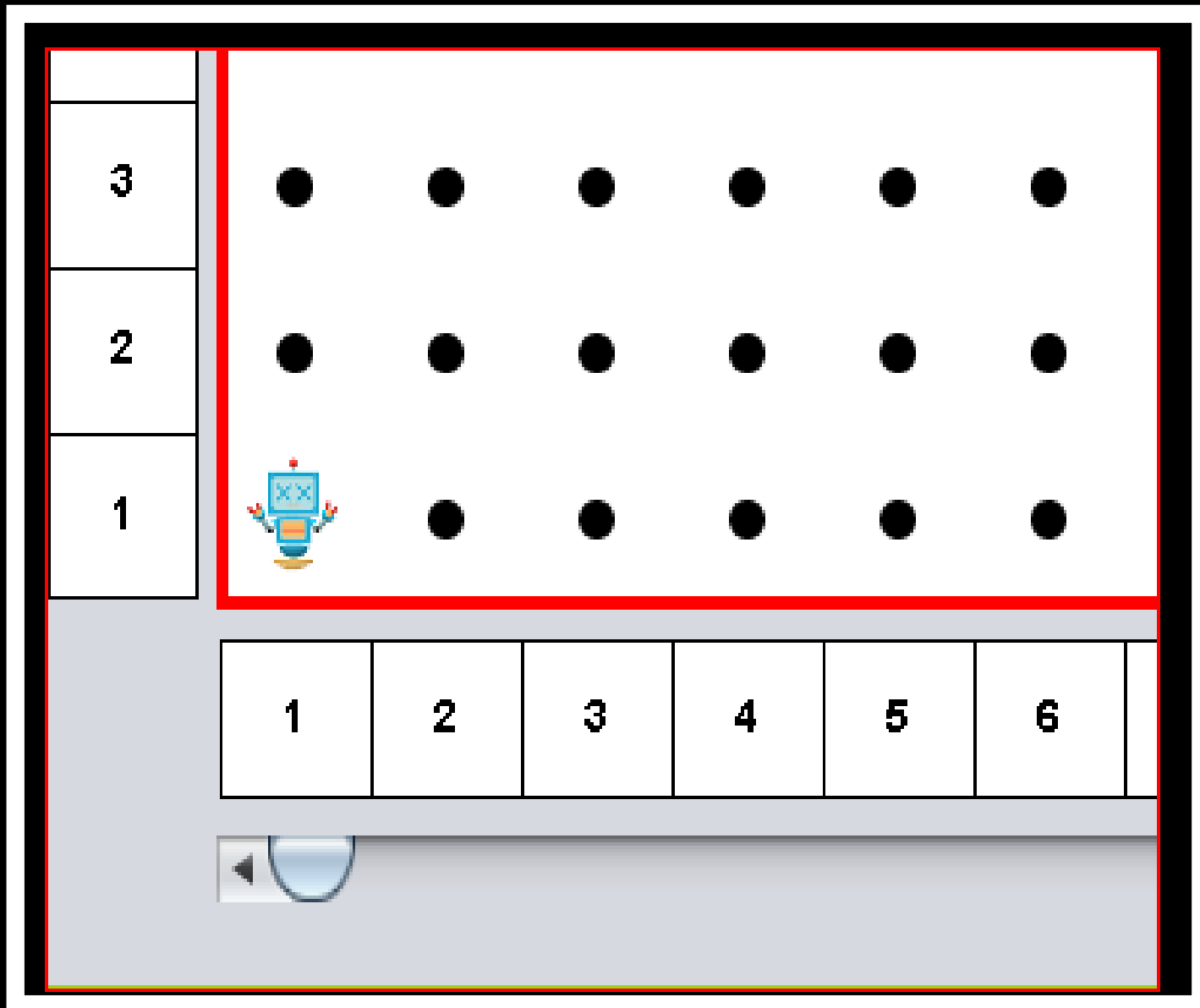
What happens if  
we program  
Thunder to move  
when a wall is  
blocking  
movement?

```
# given Thunder at 1st and 4th  
# facing south  
move  
move  
move  
move  
move
```



An impossible  
situation . .

Why? The wall is  
impenetrable – the robot is  
destructible!



Thunder has  
exploded!!

Well . . . at least its brain  
has!

- Each Thunder the Robot instruction must be on a separate line. A sequence of one or more Thunder the Robot instructions that **are all indented the same number of spaces** compose a **block** of code.
- **<instruction>** refers to any of the five primitive instructions, conditional branching or iteration instructions, or a user defined instruction.
- Thus, a block appears as:

**<instruction>**

**<instruction>**

**...**

**<instruction>**

## Control Structures

Sequence or “block”

- Conditional branching refers to the ability of a program to alter it's flow of execution based on the result of the evaluation of a conditional. The three types of conditional branching instructions in Thunder the Robot are `if` and `if/else` and `if/elif/else`. In the below examples, `<test>` refers to one of the eighteen conditionals or sensor tests presented previously. Each `<test>` evaluation results in true or false.

```
if <test>:  
    <block>
```

```
if <test>:  
    <block>  
else:  
    <block>
```

```
if <test>:  
    <block>  
elif <test>:  
    <block>
```

```
...  
elif <test>:  
    <block>  
else:  
    <block>
```

## Control Structures

Conditional or selection

NOTE that the controlled blocks are “opened” by placing a colon (:) immediately after the `<test>`

We say that the colon “opens” a scope of control. Yes, this is very Python like (by design)

- Iteration refers to the ability of a program to repeat an instruction (or block of instructions) repeatedly. The two types of iteration instructions are the definite and indefinite.
- Definite iteration is supported by the **do** **<pos\_number>** where **<pos\_number>** must be an unsigned integer greater than zero.

```
do <pos_number>:  
    <block>
```

- Indefinite iteration means that the number of iterations can vary from none to many with termination controlled by some **<test>**.

```
while <test>:  
    <block>
```

- **NOTE:** that the do repetition is the only place in the Thunder the Robot language where counting is possible.

## Control Structures

Iteration  
repetition  
looping

Again note the colon ( : )  
“opens” the scope of  
control

```
*Code editor  *World editor

1 #####
2 # FILE:  program02.thunder
3 #
4 # Purpose:  With Thunder facing east, have Thunder
5 #           traverse a five by five square course,
6 #           return to the starting position and
7 #           turnoff
8 #####
9 do 4:
10     move
11     move
12     move
13     move
14     move
15     turnleft
16 # end do
17
18 turnoff
19
```

# Using definite repetition

And adding comments to enhance readability

See program02.thunder

Comments? The symbol # begins a comment with the remainder of the line considered the comment. Comments are ignored by the interpreter.

This program assumes the world configuration of program01.world

code editor

\*World editor

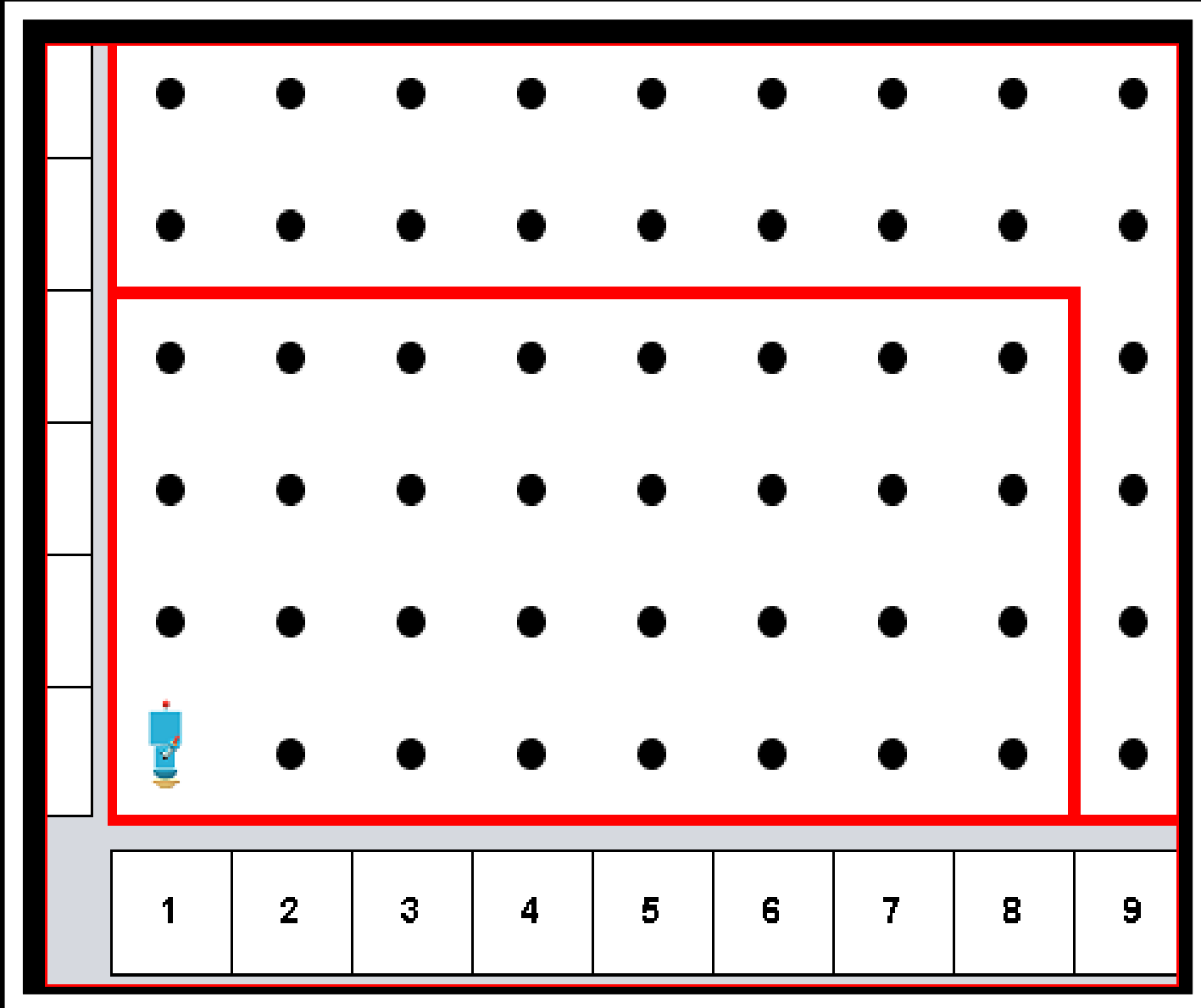
```
#####  
# FILE:  program03.thunder  
#  
# Purpose:  With Thunder facing east, have Thunder  
#           traverse a five by five square course,  
#           return to the starting position and  
#           turnoff  
#####  
  
do 4:  
  do 5:  
    move  
  #end do  
  turnleft  
#end do  
  
turnoff
```

## And refining a bit more

See `program03.thunder`

This program also assumes the  
world configuration of  
`program01.world`





## Using indefinite iteration

Yes, if we knew that the boundaries were fixed and would remain the same, we could solve this similarly to the five by five.

We can simply edit our previous program to fit the new configuration. However, we have to edit our program for each different world configuration. We would like to arrive at a more general solution that would work for any rectangular world configuration.

ode editor \*World editor

```
#####  
# FILE:  program04.thunder  
#  
# Purpose:  With Thunder facing east, have Thunder  
#           traverse a rectangular enclosure,  
#           return to the starting position and  
#           turnoff  
#####  
do 4:  
    # handle an arbitrary number of moves  
    while front_is_clear:  
        move  
    #end while  
    turnleft  
# end do  
  
turnoff
```

## Using indefinite iteration

We have a problem similar to the previous one EXCEPT that we now have constraints (walls) that we cannot penetrate AND we do not know in advance how big the enclosure is.

See program04.thunder

This program assumes the world configuration of program04.world

\*Code editor

World editor

```
1 #####
2 # FILE:  program05.thunder
3 #
4 # Purpose:  With Thunder facing east, have Thunder
5 #           traverse a rectangular enclosure of
6 #           arbitrary size and stop when a beeper
7 #           is found.  Then Thunder should turnoff.
8 #           Assume that the beeper is located at the
9 #           beginning of one side of the enclosure
10 #####
11 move
12 while not_next_to_a_beeper:
13     # handle an arbitrary number of moves
14     while front_is_clear:
15         move
16     #end while
17     turnleft
18 # end while
19
20 turnoff
21
```

## Using indefinite iteration

Now lets change the problem up just a bit. We want Thunder to walk around the enclosure until a beeper is found. Then Thunder is to turnoff. Assume that the beeper is at the beginning of one of the sides of the enclosure.

See `program05.thunder` and `program05.world`

```

3  #
4  # Purpose:  With Thunder facing east, have Thunder
5  #           traverse a rectangular enclosure of
6  #           arbitrary size and stop when a beeper
7  #           is found.  Then Thunder should turnoff.
8  #           Assume that the beeper is located at the
9  #           beginning of one side of the enclosure.
10 #
11 #           Traversal is accomplished clockwise.
12 #####
13 define turnright:
14     do 3:
15         turnleft
16     #end do
17 # end turn_right
18
19 move
20 while not_next_to_a_beeper:
21     # handle an arbitrary number of moves
22     while front_is_clear:
23         move
24     #end while
25     turnright
26 # end while
27
28 turnoff

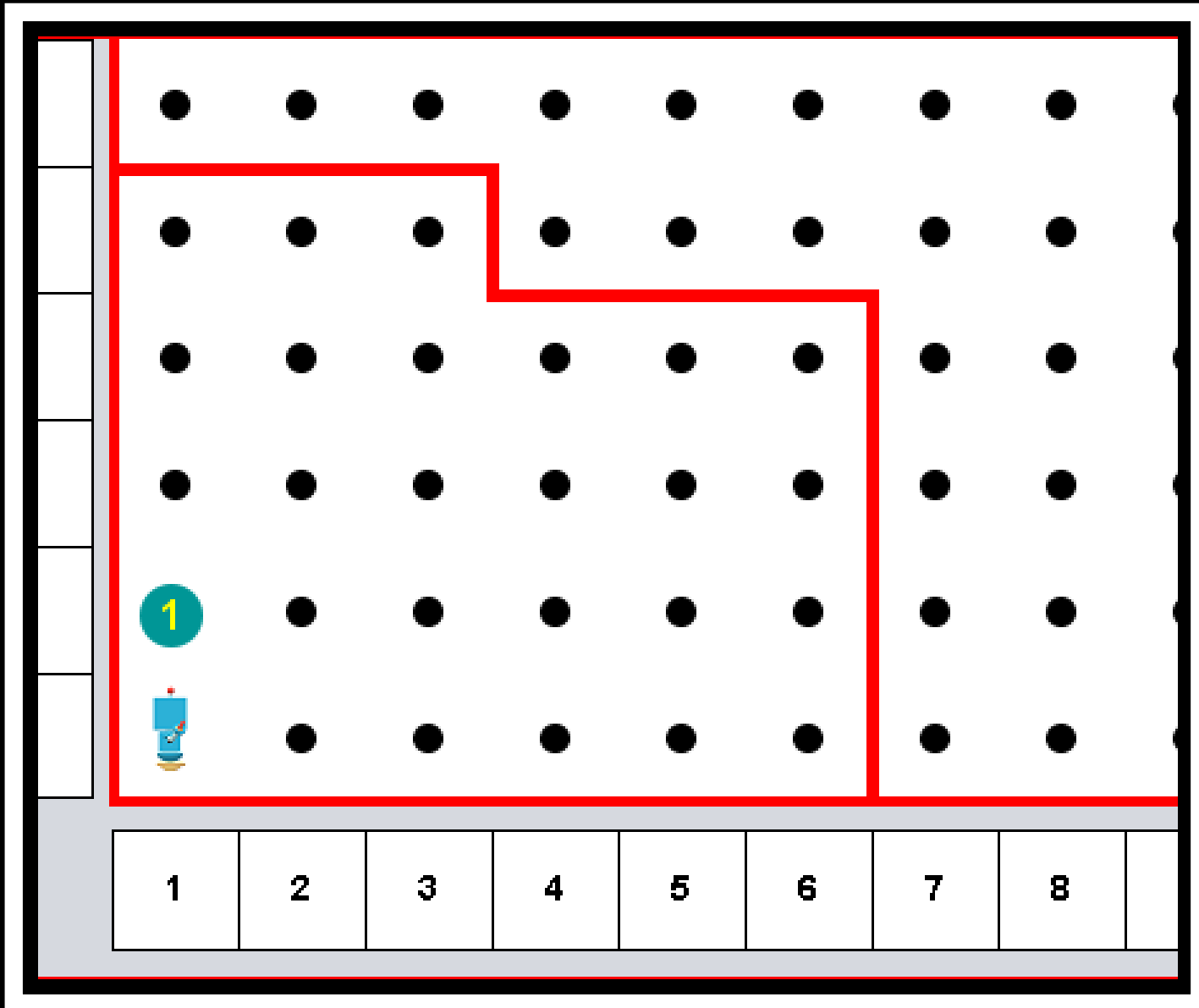
```

## Introducing procedural abstraction

Now let's change the problem up just a bit. We want Thunder to walk around the enclosure until a beeper is found. Then Thunder is to turnoff. Assume that the beeper is at the beginning of one of the sides of the enclosure. However, instead of completing the course counter clockwise, we want to navigate the course **clockwise**.

Problem!! Thunder does not know how to turn right. We can define a new instruction, however.

See `program06.thunder` and `program06.world`



## A challenging problem

Navigate an enclosure that has irregular shape and stop when on top of a beeper.

See `program07.world`

Issues:

- Cannot collide with a wall
- Must navigate around walls that fall away from us

Strategy:

- How do we follow the wall?
- One strategy is to always follow the wall to the right



See program07a.world

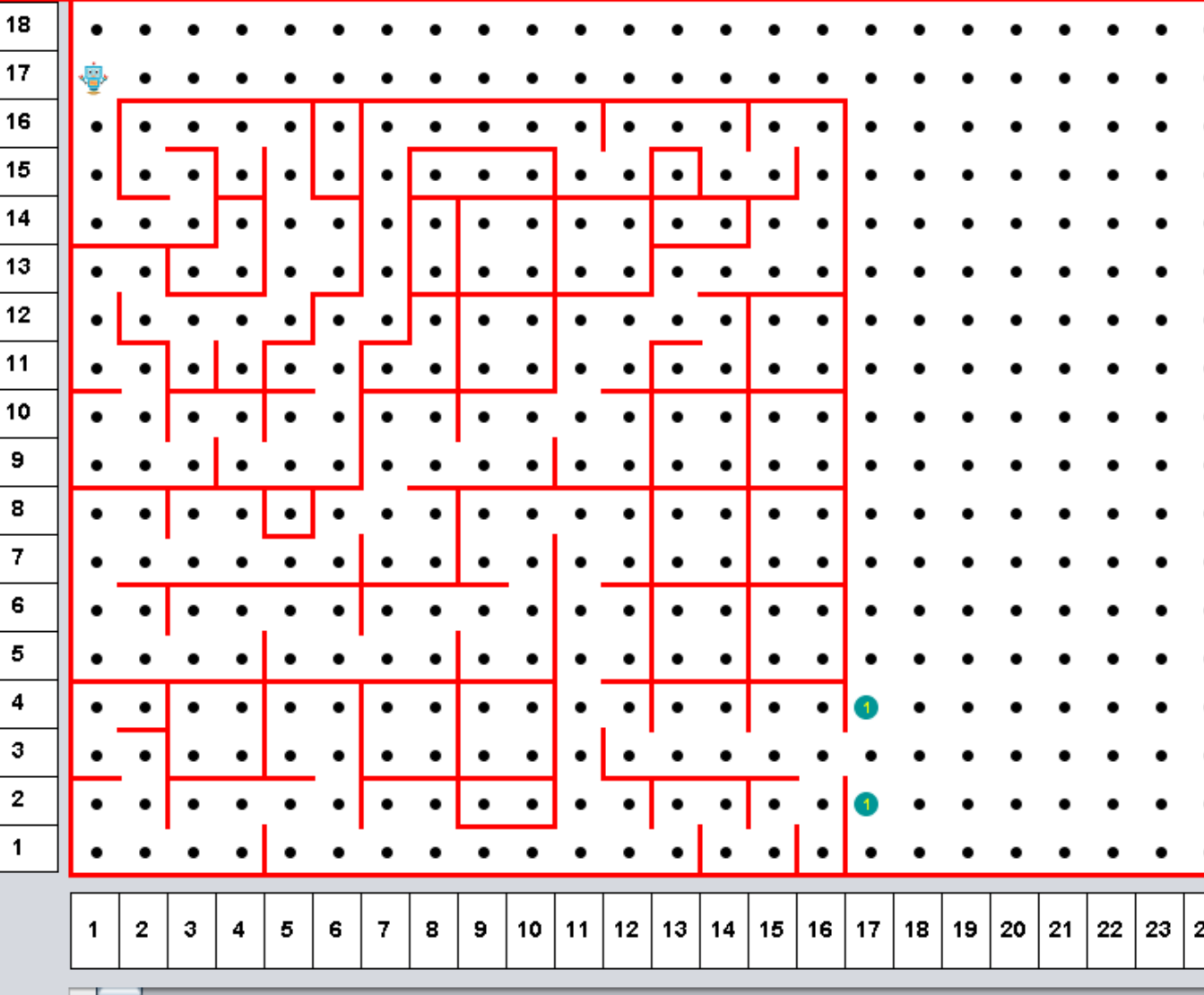
```

1 #####
2 # FILE:  program07.thunder
3 #
4 # Purpose:  With Thunder facing east, have Thunder
5 #           traverse a rectangular enclosure of
6 #           arbitrary size and stop when a beeper
7 #           is found.  Then Thunder should turnoff.
8 #           Assume that the beeper is located at the
9 #           beginning of one side of the enclosure.
10 #
11 #           Traversal is accomplished counter
12 #           clockwise.
13 #####
14 define turnright:
15     do 3:
16         turnleft
17     #end do
18 # end turn_right
19
20 def follow_wall_right:
21     if right_is_clear:
22         turnright
23         move
24     elif front_is_clear:
25         move
26     else:|
27         turnleft
28     #end if
29 #end follow_wall_right
30
31 move
32 while not_next_to_a_beeper:
33     follow_wall_right
34
35 turnoff

```

Follow wall  
to the right

See  
program07.thunder



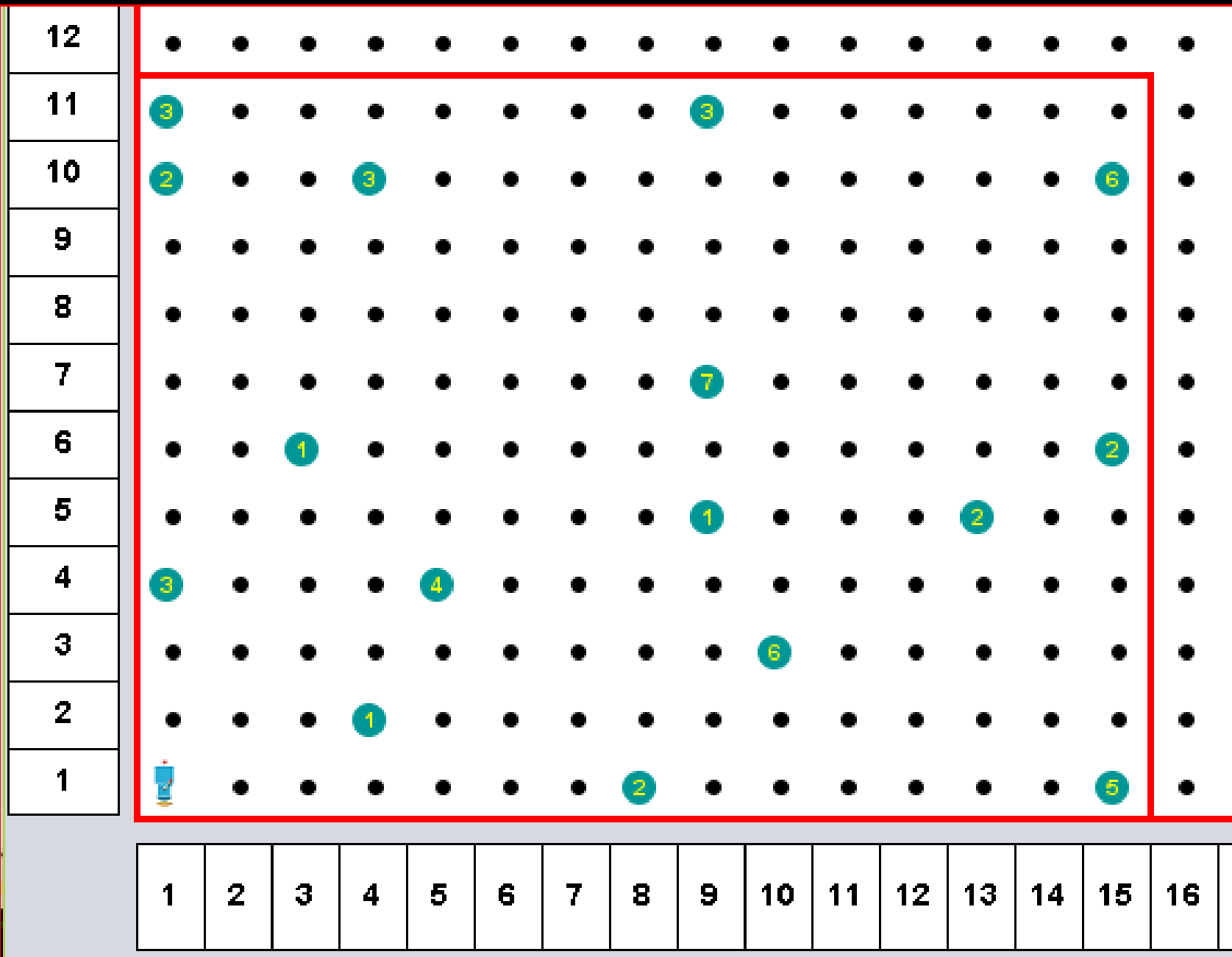
`follow_wall_right`  
good for  
navigating a  
maze

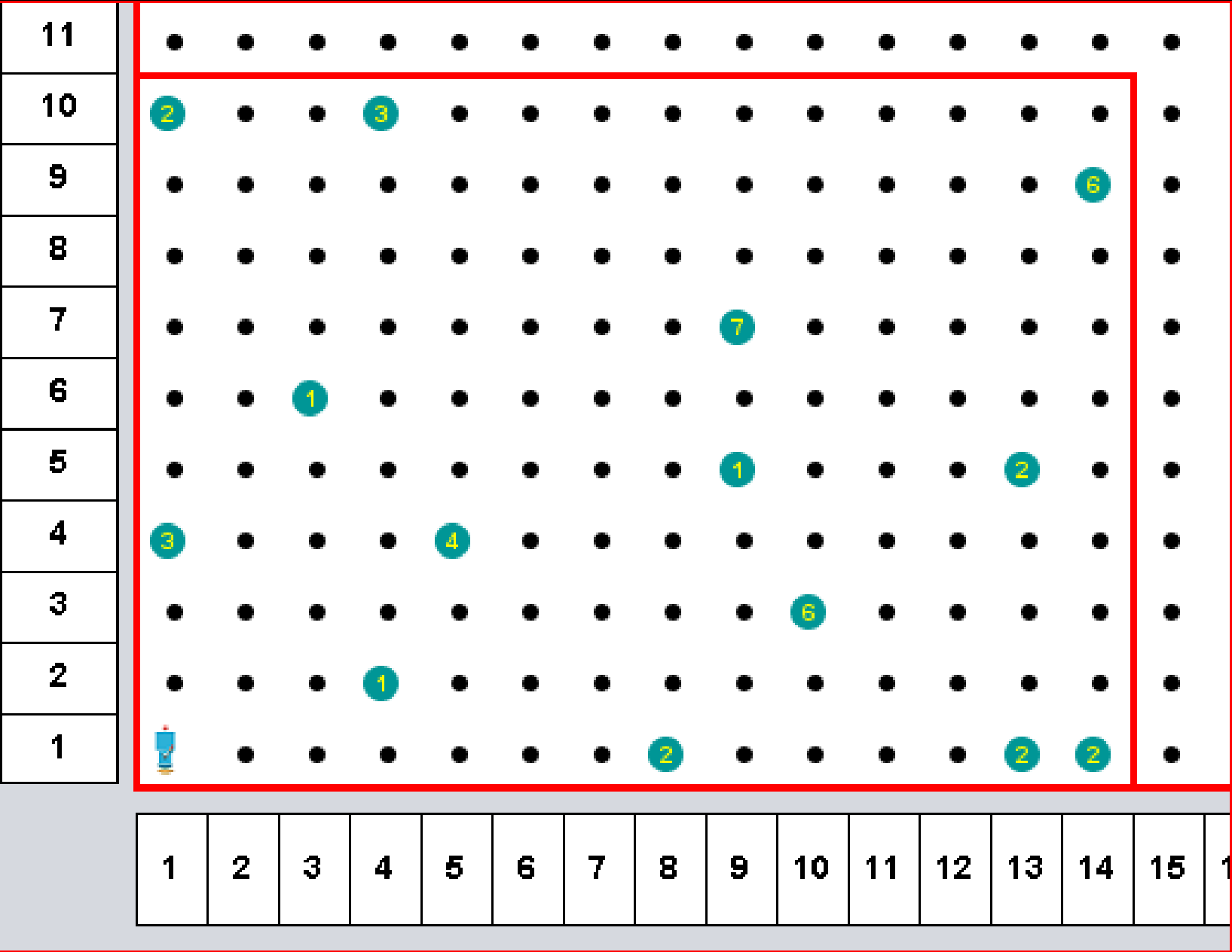
- Assume that there are NO islands in the maze
- Assume that we place a beeper to the right of the single exit from the maze
- Assume that we position Thunder so that it is at the entrance
- See `mazeBig.world`



# How about cleaning trash from an enclosed park?

- An enclosed “park” of arbitrary length and arbitrary width (avenues and streets) contains scattered piles of beepers. It is Thunder’s responsibility to start in the southwest corner of the park facing east and sweep by street and clean up all the piles of beepers.
- Once all the beepers are cleaned up, they are to be deposited in the northeast corner of the park.
- Then Thunder is to return to the southwest corner, face east and turn off.
- See `cleanTrash.thunder` and `trash1.world` and `trash2.world`

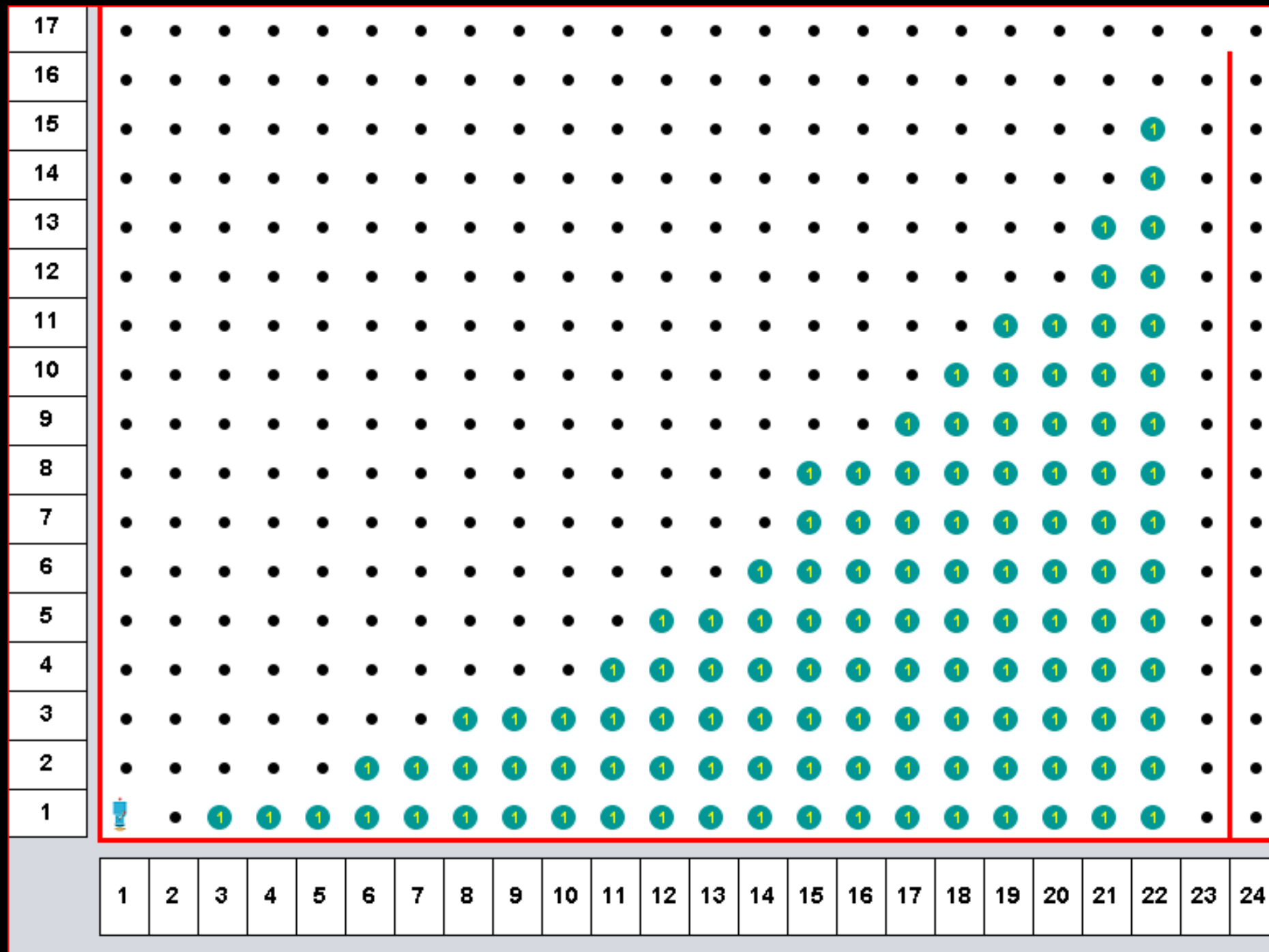




# How about “sorting” into ascending order the heights of columns of beepers

- See `Sort.thunder` and `Sort.world`
- Givens:
  - “stacks” of beepers of arbitrary height from zero to many
  - Each vertical position of the “stacks” is represented by a single beeper at an intersection
  - Arbitrary number of “stacks”
  - “stacks” are bounded on east and west by a wall that is at minimum at least one unit higher than the height of the tallest “stack” of beepers
  - “stacks” of beepers are guaranteed to exist in a situation in which there is an empty “stack” on the west and the east of the “stacks” to be considered
  - Thunder is to start the sort by facing east at 1<sup>st</sup> Avenue and 1<sup>st</sup> Street and is to return to the “home” position at 1<sup>st</sup> Avenue and 1<sup>st</sup> Street.
  - Solution was developed under constraint that the “main” logic is to drive what happens including turning off.





# Iteration vs. Recursion

## Iteration

```
def face_east:
  while not_facing_east:
    turnleft
  #end while
# end face_east
```

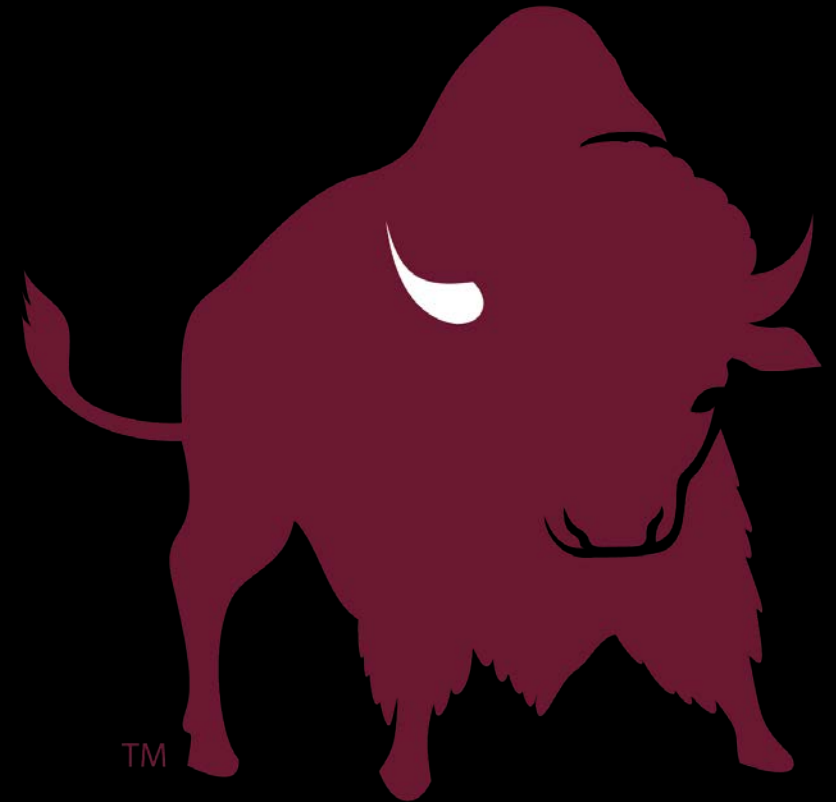
## Recursion

```
def face_east:
  if not_facing_east:
    turnleft
    face_east
  #end if
# end face_east
```

Questions, observations, etc.



# The End



TM