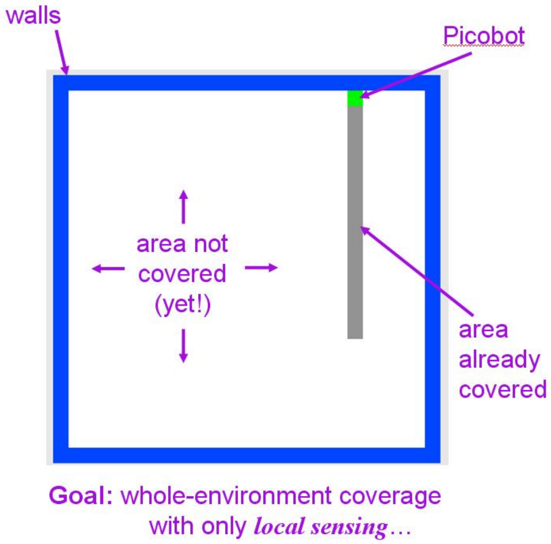
COSC 101 Picobot Reference

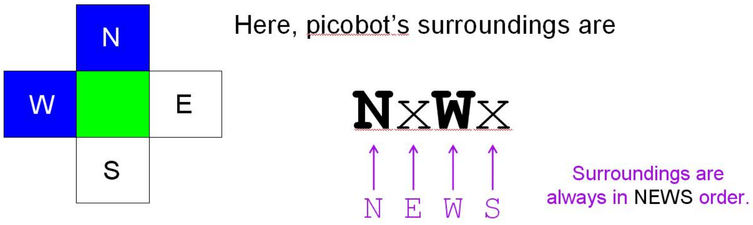
The Picobot language was presented in class. What follows is mainly a reference, plus some tips on completing the maze room.

Picobot starts at a random location in a room -- you don't have control over Picobot's initial location. The walls of the room are blue; Picobot is green, and the empty area is white. Each time Picobot takes a step, it leaves a grey trail behind it. When Picobot has completely explored its environment, it stops automatically.



Surroundings

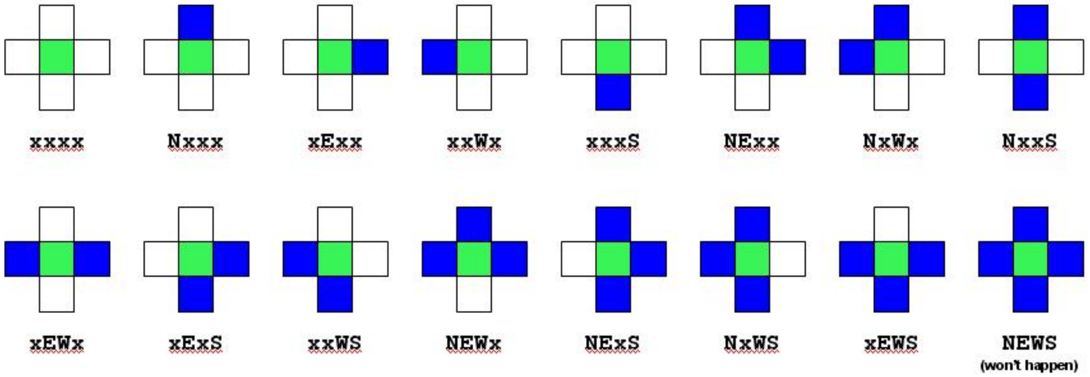
Not surprisingly, Picobot has limited sensing power. It can only sense its surroundings immediately to the north, east, west, and south of it. For example,


N
W
E
S
Here, picobot's surroundings are
NxWx
N E W S
Surroundings are always in NEWS order.

In the above image, Picobot sees a wall to the north and west and it sees nothing to the west or south. This set of surroundings would be represented as follows:

```
NxWx
```

The four squares surrounding Picobot are always considered in NEWS order: an x represents empty space, the appropriate direction letter (N, E, W, and S) represents a wall blocking that direction. Here are all of the possible Picobot surroundings:



State

Picobot's memory is also limited. In fact, it can store only a single number from 0 to 99. This number is called Picobot's **state**. In general, "state" refers to the relevant context in which computation takes place. For example, an app on a mobile phone might maintain state such as the current location (available via GPS) and the speed (available via the accelerometer). Here, you might think of the Picobot's "state" as a way in which Picobot can keep track of its progress towards achieving its overall goal.

Picobot always begins in state 0.

The state and the surroundings are all the information that Picobot has available to make its decisions!

Rules

Picobot moves according to a set of rules of the form:

```
StateNow   Surroundings   ->   MoveDirection   NewState
```

For example,

```
0          xxxS           ->   N               0
```

is a rule that says "if Picobot starts in state 0 and sees the surroundings xxxS, it should move North and stay in state 0."

The MoveDirection can be N, E, W, S, representing the direction to move. The MoveDirection can also be X, which means do not move. This can be handy when you only want Picobot to change to a new state.

A Picobot program is simply a list of rules.

How Picobot moves

Picobot follows a very simple algorithm to move around the room. First, it looks up its state. Then it senses its surroundings. Then it consults its program, which is just a list of rules. When it finds a matching rule -- meaning that StateNow of the rule matches Picobot's state and the rule's Surroundings matches Picobot's current surroundings --

then it executes that rule. Executing the rule may cause Picobot to move and possibly to change state. The process repeats until either (a) the room is covered, or (b) Picobot finds itself in a situation where no rule matches.

Suppose the above rule were Picobot's only rule. In addition, suppose Picobot began in state 0 at the bottom of an empty room, it would move up (north) one square and stay in state 0. However, **Picobot would not move any further**, because its surroundings would have changed to xxxx, which does not match any rule in its program.

Wildcards

The asterisk `*` can be used inside surroundings to mean "I don't care whether there is a wall or not in that position." For example, `xE**` means "there is no wall North, there is a wall to the East, and there may or may not be a wall to the West or South."

As an example, the rule

```
0   x***   ->   N   0
```

is a rule that says "if Picobot starts in state 0 and sees any surroundings without a wall to the North, it should move North and stay in state 0."

If this new version (with wildcard asterisks) were Picobot's only rule and if Picobot began (in state 0) at the bottom of an empty room, it would first see surroundings `xxxS`. These match the above rule, so Picobot would move North and stay in state 0. Then, its surroundings would be `xxxx`. These also match the above rule, so Picobot would again move North and stay in state 0. In fact, this process would continue until it hit the "top" of the room, when the surroundings `Nxxx` no longer match the above rule.

Example program

The following Picobot program can be used to cover a "pencil" room (i.e., a "tall" room that is as "skinny" as possible, only one block wide).

```
# state 0 goes N as far as possible
0 x*** -> N 0   # if there's nothing to the N, go N
0 N*** -> X 1   # if N is blocked, switch to state 1

# state 1 goes S as far as possible
1 ***x -> S 1   # if there's nothing to the S, go S
1 ***S -> X 0   # otherwise, switch to state 0
```

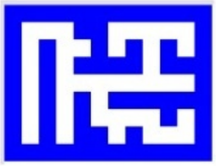
Anything after a pound sign (`#`) on a line is a comment. (This is true in python as well.) Comments are human-readable explanations of what is going on, but ignored by Picobot. Blank lines are ignored as well.

Recall that Picobot always starts in state 0. Picobot now consults the rules from top to bottom until it finds the first rule that applies. It uses that rule to make its move and enter its next state. It then starts all over again, looking at the rules and finding the first one from the top that applies.

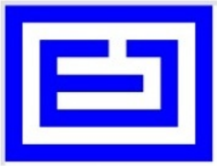
In this case, Picobot will follow the first rule up to the "top" of its environment, moving north and staying in state 0 the whole time. Eventually, it encounters a wall to its north. At this point, the topmost rule no longer applies. However, the next rule `0 N*** -> X 1` does apply now! So, Picobot uses this rule which causes it to stay put (due to the `X`) and **switch to state 1**. Now that it is in state 1, neither of the first two rules will apply. Picobot follows state 1's rules, which guide it back to the "bottom" of its environment. And so it continues.

Solving the maze

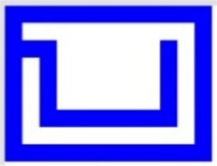
Getting Picobot to cover a maze, such as the one shown in the Picobot simulator, can be both fun and challenging. The maze in the simulator has the following property: all the walls are connected to the outer boundary and all empty cells are adjacent to a wall. A smaller maze with this property is shown in (a) in the figure below. Any maze with this property can be completely explored with a simple algorithm called the right-hand rule (or the left-hand rule if you prefer).



(a)



(b)



(c)

Imagine for a moment that you are in the maze rather than Picobot. In contrast to Picobot, you have a clear sense of the direction you're pointing and you have two hands. You start facing north with your right hand touching the wall. Now, you can visit every empty cell by simply walking through the maze, making sure that your right hand is always touching the wall. Pause here for a moment to convince yourself that this is true. Notice also that this algorithm will not visit every cell if some walls are not connected to the outer boundary, as shown in the maze in (b) or if some empty cells are not adjacent to a wall, as shown in (c).

It is important that you understand the right-hand rule before thinking about how to program it in Picobot. You might find it helpful to see the rule in action. Here is one of many youtube videos showing the rule in action: `http://youtu.be/AoCeJxvVrUU`. Pause the video and try to predict where the robot should move next.

Converting the right-hand rule into a set of Picobot rules is an interesting computational challenge. After all, you have a sense of direction and you have a right hand that was guiding you around the walls, whereas Picobot has neither hands nor a sense of orientation. To "teach" Picobot the right-hand rule, we can use states to represent the direction that Picobot is pointing. It may seem that an impossibly large number of situations must be considered, but in fact, the number of situations is finite and actually quite small, which makes it possible to program Picobot for this task.

Here's the idea. The challenge is that the right-hand rule describes actions in "relative" terms -- go straight, turn right, etc. -- whereas Picobot rules are expressed in absolute terms -- move north, move east, etc. To overcome this challenge, we can rely on Picobot's state. Suppose we (arbitrarily) choose state 0 to correspond to representing Picobot pointing north. Picobot's imaginary right hand is then pointing east. If there is a wall to the east and none to the north, this means Picobot's hand is on the wall and there is nothing in front of it. The right-hand rule would tell us to "keep going straight." Translating "go straight" to Picobot means to move one step to the north and keep pointing north. Taking a step to the north is no problem. "Keep pointing north" means "stay in state 0." On the other hand, if we are in state 0 and there is suddenly no wall to the east, then this means Picobot's hand "fell off" the wall. To get Picobot's hand back on the wall, Picobot should turn right and step forward. How do we translate this idea into Picobot language? Well, since Picobot is in state 0, it's facing north. Thus, "turning right" would mean turning to face East. This means changing to a different state, a state that is intended to correspond to representing Picobot pointing east. Having turned east, taking a step forward means taking a step east. To complete the program, first think about what other situations Picobot might encounter. For example, what if Picobot is facing north and there is a wall to the north? Remember, your program should work regardless of where Picobot starts.

---

[*] Acknowledgments: Picobot has been adapted with permission from an assignment by Zach Dodds and Wynn Vonnegut.