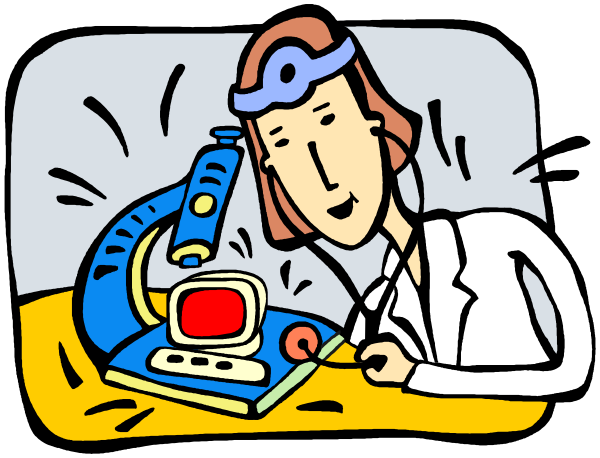
Welcome to CS 42

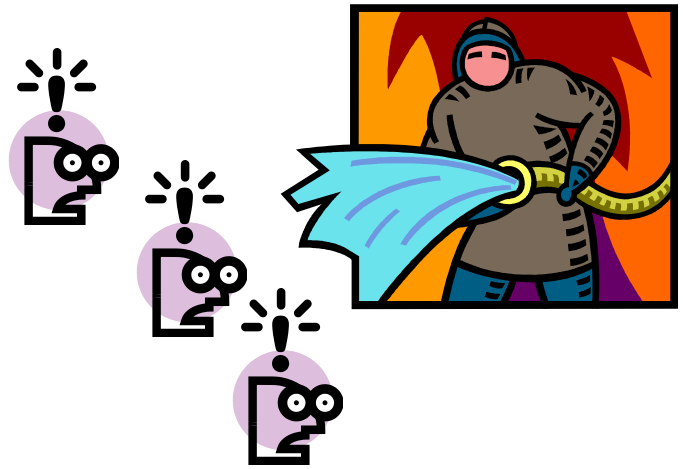
Principles and Practice of Computer Science



Interactions & properties

- matter and energy — physics
- molecules — chemistry
- cells and organisms — biology
- rule-based systems — mathematics
- information — computer science

42 ≈ 5 + 60



- You are in CS 42 because you have significant CS experience.
- If you feel you would be better placed in CS 5, talk to me ASAP.
- CS 42 combines the best of CS 5 and CS 60, but moves faster.
- You get credit as if having taken CS 60 and 5.
- CS 42 prepares you for:
  - Any course with CS 60 as a prerequisite
  - Future computational work
  - A happy and fulfilling life :-)

Important Addresses

1600 Pennsylvania Ave. NW, Washington DC 20006

Professor Keller's Office:

1253 Olin Building (1250 N. Dartmouth Ave.)

Home page:

`http://www.cs.hmc.edu/courses/current/cs42`

Email for help:

`cs42help@cs.hmc.edu`

Homework submission site:

`http://www.cs.hmc.edu/~submissions/submissionsFall12/`

Picobot: `http://www.cs.hmc.edu/picobot/`

(((Racket))): `http://racket-lang.org/download/`

Other Administrivia

Lectures **Tuesday, Thursday:** 1:15-2:30 pm

Key skills, topics, and their motivation

Insight into the HW problems (what, ***why***, how)

Required! Let me know if you won't make it.

Tutoring and Office Hours http://www.cs.hmc.edu/courses/2012/fall/cs42/tutorhours.html

Monday, Tuesday, Wednesday: 2:45-4:00 pm

Can't make these hours? Just contact me by email: keller@cs.hmc.edu

Homework **Monday nights:** due by 11:59 pm

Grading

Grades

Based on points percentage

~ 60% Assignments

~ 30% Exams

~ 10% Participation/"quizzes"

```
if perc >= .95: grade = 'HP'
elif perc >= .70: grade = 'P'
```

Exams

(page of notes OK)

Midterm	TB Scheduled
Final	TB Scheduled

Note!

To pass CS 42, you ***must*** have a passing grade on each of the three individual components of the course (exams, HW, and participation/quizzes). Failing one component results in failing CS 42, even with a passing total score.

Homework Policy

Assignments

~ 4 problems/week ~ 100 points extra credit available

Due Monday evening - by 11:59 pm. You have 3 ***CS 42 Euros*** to use…
"Late Days"

Collaboration

Some problems are specified "individual-only."

Others offer the option of working in a pair.

- You don't have to
- If you do, you must share the work equally - typing and coaching
- You should make ONLY ONE submission for both of you
- Be sure to indicate who your partner was at the submission site!
- You may want to email your partner the code, too.

Honor Code →

Honor Code

- You are *encouraged* to **discuss** problems with other students – or tutors - or the instructor.

- You may **not share** written, electronic or verbal solutions with other students (present or past):
  - No internet or other copying of files except those provided by the course material.
  - No transcribing of programs from paper, whiteboards, blackboards, or other media.

You will have the option of working in pairs for MANY of each week's problems: the same guidelines apply for each pair.

Read over the syllabus and honesty policy – sign & submit today!

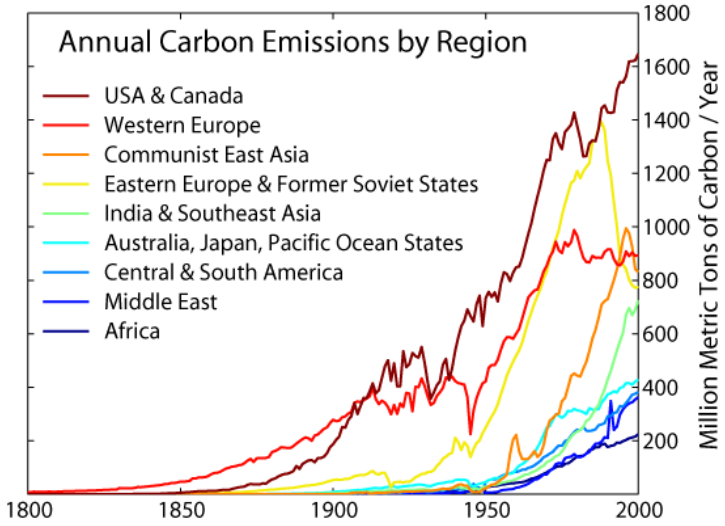
Lecture Slides

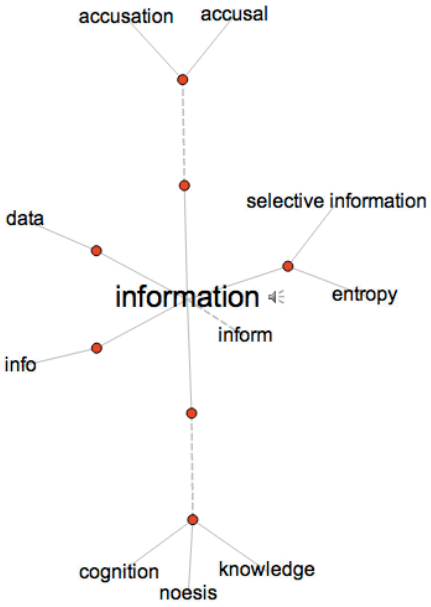
- It is best if you **take notes**. It helps your memory and focus. Also, some things will be on the board only.

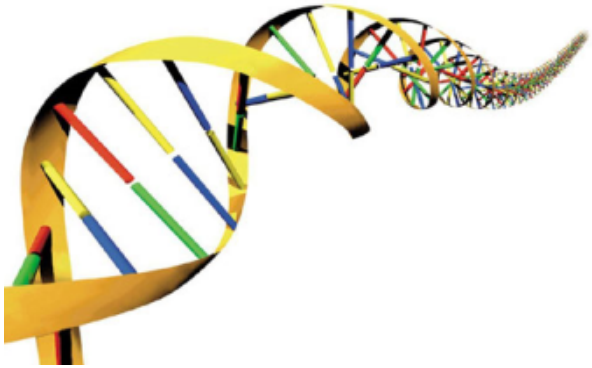
- I will post slides, when used (usually before the lecture, but no guarantees).

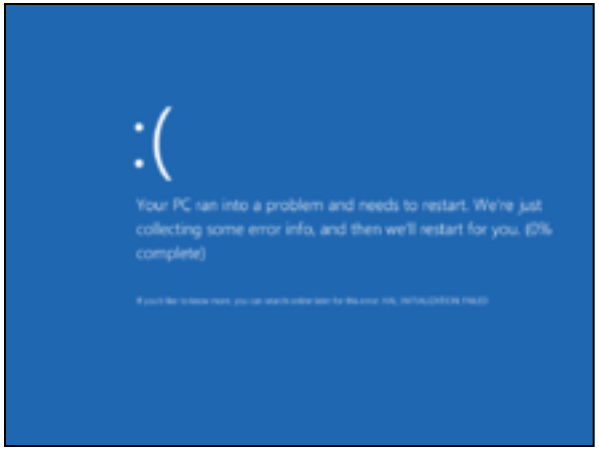
- I will not print them out (but you may if you wish).

What is Information?









Information

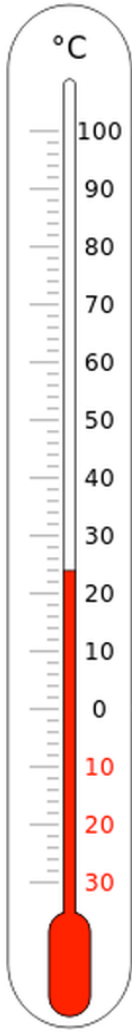
Information is a sequence of symbols, or signals, upon which a **decision** can be based.

Digital vs. *Analog* Information

Digital information is based on **discrete** units, units that can be *counted*.

Analog information is based on **continuous** units, units that must be *measured*.

[What is the source of the word “analog”?]



Digital vs. *Analog* Computers

IBM 1620 Computer, 1960's

- Punched card, paper tape and keyboard input; card, paper tape and printed output.
- Simultaneous read, compute and punch when using card input-output.
- Large-capacity core storage - up to 60,000 digits.
- High internal processing speeds. Access time - 20 microseconds.

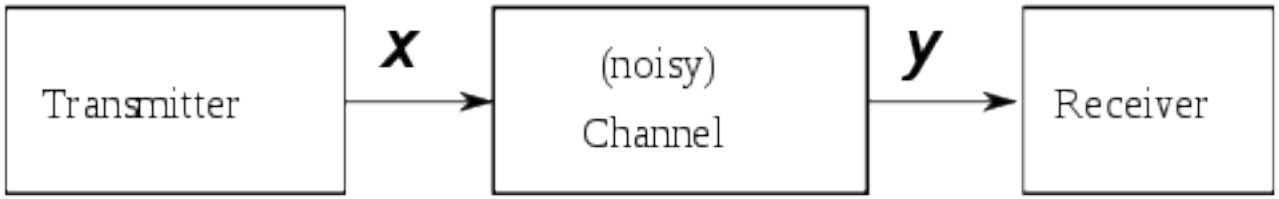
Electronic Associates TR-48
Analog Computer, 1960's

Is Computer Science “Information Theory”?

No. The term “information theory” has already been taken.

Claude Shannon, 1948, started the development of this theory, which relates to methods of sending information over a noisy channel. He called it a theory of *communication*, but others adopted the term “information theory”.

[Shannon also introduced Boolean algebra to Engineering.]



http://en.wikipedia.org/wiki/Information_Theory

Claude Elwood Shannon (1916–2001)

Is Computer Science “Information Science”?

Short answer: *No*

http://en.wikipedia.org/wiki/Information_science

Information science (or information studies) is an interdisciplinary field primarily concerned with the analysis, collection, classification, manipulation, storage, retrieval and dissemination of information. Practitioners within the field study the application and usage of knowledge in organizations, along with the interaction between people, organizations and any existing information systems, with the aim of creating, replacing, improving or understanding information systems.

Information science is often (mistakenly) considered a branch of computer science. However, it is actually a broad, interdisciplinary field, incorporating not only aspects of computer science, but often diverse fields such as archival science, cognitive science, commerce, communications, law, library science, museology, management, mathematics, philosophy, public policy, and the social sciences.

Information science should not be confused with information theory, the study of a particular mathematical concept of information, or with library science, a field related to libraries which uses some of the principles of information science.

What is Computer Science?

Computer science is the study of methods and means of **computation**, including:

Algorithms for computation

Efficiency of computation

Expressiveness of computation models

Computability

As well as aspects of:

- Information representation
- Information communication and security

What is Computation?

Computation is any process of **transforming information** from one form to another.

Ideally, such a transformation does at least one of the following:

Makes the information easier to understand or visualize.

Abstracts the information into a more succinct form.

Makes the information easier to transmit.

Uses the information to find further information.

Determines the veracity of the information.

What/When was the *first …*

operational automatic Computer?

programming language?

programmer?

Near Synonyms for Computer Science

Informatics

Artificial Intelligence

Fields Closely Related to CS

Cybernetics

Robotics

Computer Systems

Information Systems

Computer Engineering

CS 42 Topics Overview

Theoretical CS

Computability
Running time analysis
FSMs/Regular Languages

The Machine

Digital logic
Computer Architecture

Fundamental Algorithms

Sorting
Searching

Programming Paradigms

Functional Programming (Racket)
Object Oriented Programming (Java)
Logic Programming (Prolog)
Assembly Programming (HMMM)
Python (various)

Data Structures/ADTs

Stacks
Queues
Lists
Arrays
Dictionaries

Language Implementation

Grammars
Parsing

States and Transitions

- The concept of states and transitions underlie most computational systems and models at some level.

- This brief discussion will take us to our first assignment.

States and Transitions

- A **state** is a piece of information that represents possible futures.

- A **transition** is a (binary) relation on states.

Example: Solving a Puzzle

- Suppose you have a puzzle involving a sequence of manipulations.

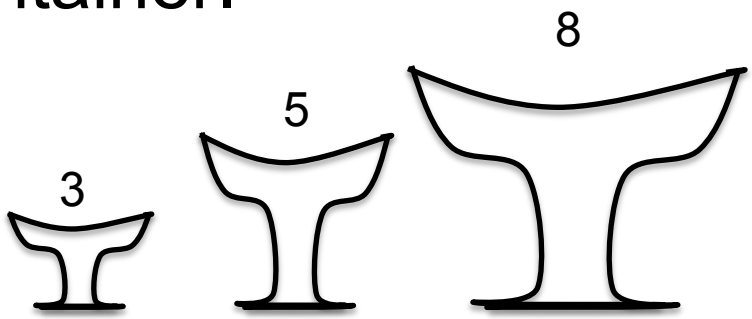
- The **state** represents what possible sequences may occur in the future.

A Water Jug Puzzle

- Given two *empty* irregular containers, 3 and 5 liters, and an 8 liter *full* container, obtain, by a sequence of pourings, exactly 4 liters in the largest container and 4 in the second-largest.

- State = What's in each container.

- Transition = Pouring.



- "irregular" means that you can't estimate the amount being poured.

State Representation

- Each state can be a triple, (C8, C5, C3), where C8 means the contents of the 8 liter container, etc.
- Transitions are possible pourings.
- Starting state is (8, 0, 0).
- Ending state is (4, 4, 0).

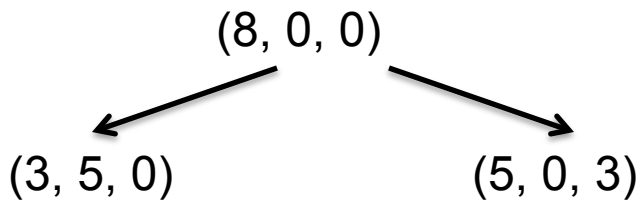
Transitions

(8, 0, 0)

???

(4, 4, 0)

Transitions

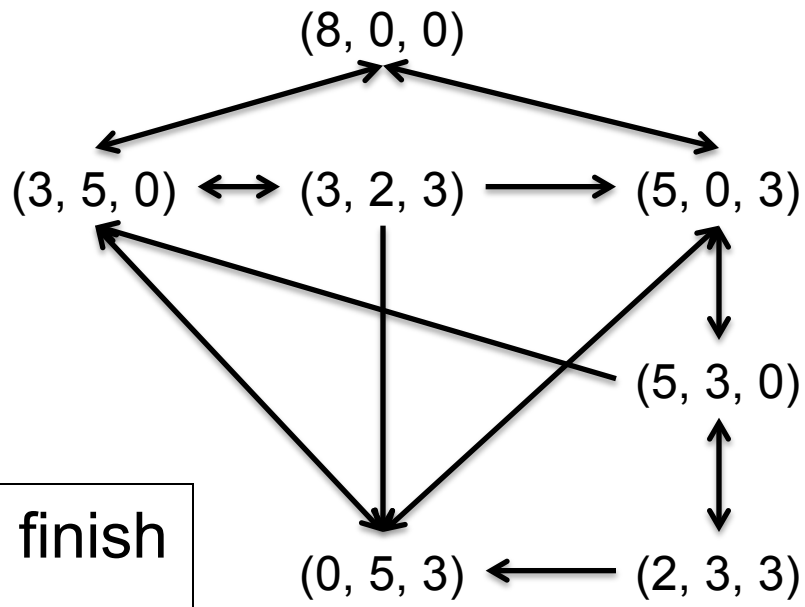


???

(4, 4, 0)

Transitions

Note that some transitions are "reversible", others not.



See if you can finish the puzzle.

How many pourings are required?

Note that keeping track of states is **essential** to ensure a solution is found if one exists, or to establish that no solution exists.

(4, 4, 0)

Computer Science

- Rather than focusing on a single problem, computer science prefers to develop **algorithms** for entire **families** of problems.

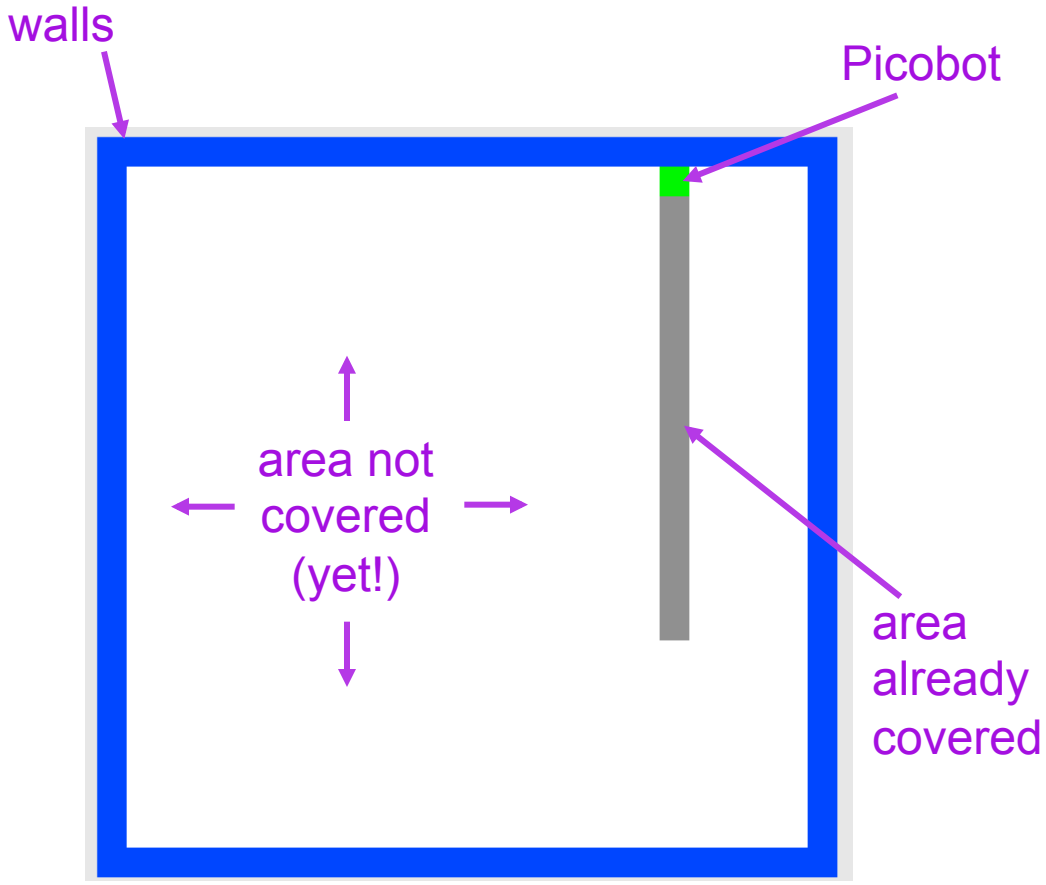
- There may be **constraints** on such algorithms, such as the number of steps (running time) or amount of memory. [complexity]

- Barring finding a solution, computer science may try to determine that **no solution exists**. [unsolvability]

Assignment 0 (also for CS 5)

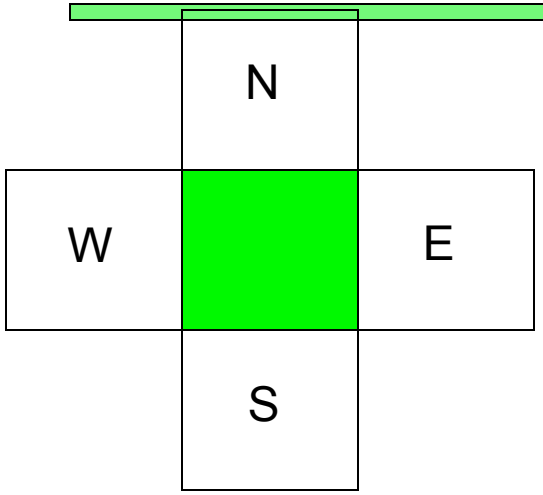
Picobot

inspiration?



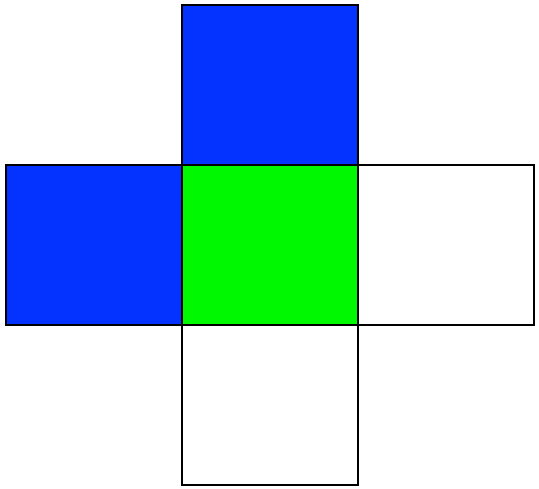
Goal: whole-environment coverage with only ***local sensing…***

Picobot Sensing



Picobot can only sense walls directly to the N, E, W, and S.

Area covered already is ***not*** sensed by Picobot (one of the things that makes this problem harder than you might expect).



For example, here its surroundings are

NxWx

N E W S

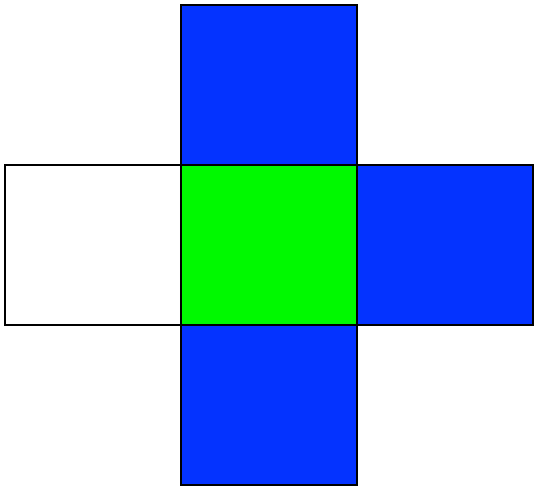
Surroundings are always in NEWS order.

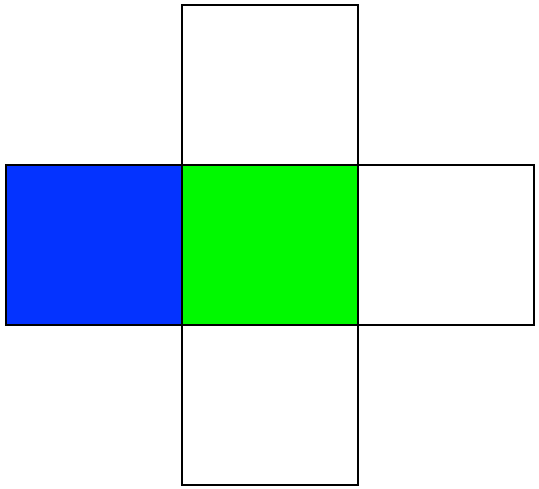
N, E, W, or S mean there is a wall.

x means no wall. * means either ("don't care)

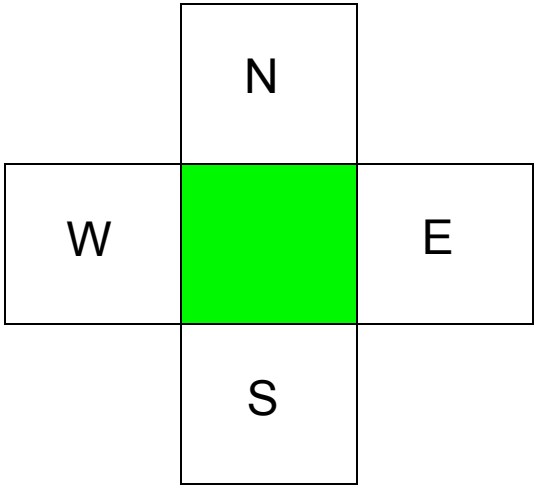
Information about Surrounding

Describe these surroundings



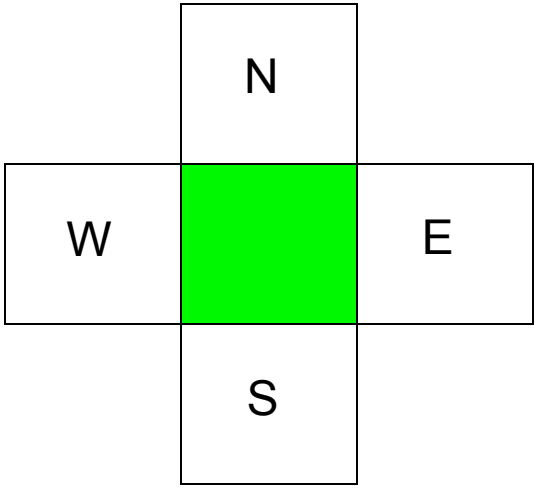


Information about Surrounding



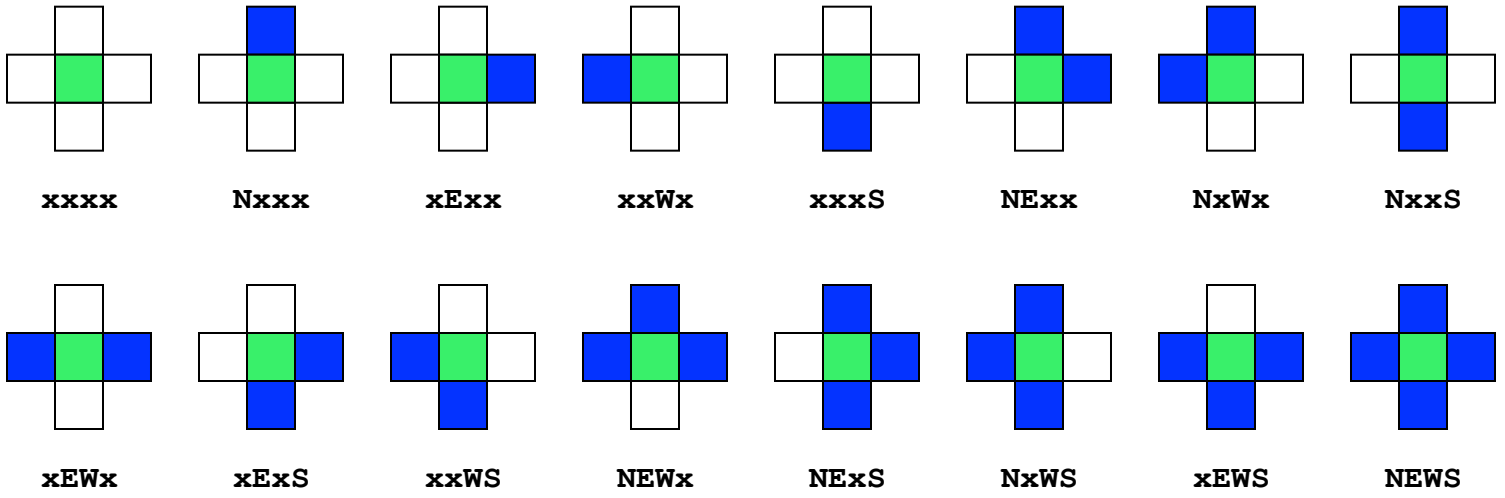
How many distinct surroundings are there?

Information about Surrounding

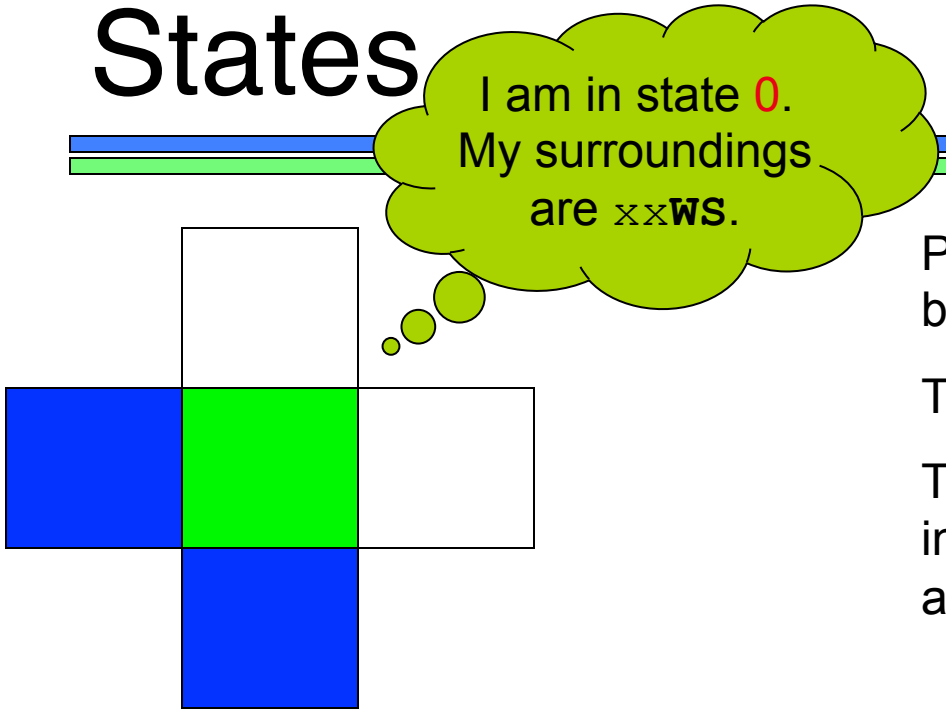


How many distinct surroundings are there?

2^4 == 16 possible …



States



Picobot's memory is a single number, between 0 and 100, called its state.

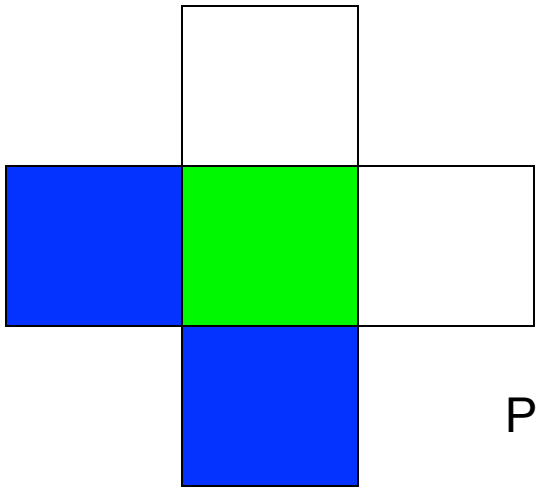
This state is the ***internal state*** of Picobot.

The ***complete state*** consists of the internal state, Picobot's position, the grid, and what has been covered.

State and immediate surroundings represent all the robot knows about the complete state.

Picobot ***always*** starts in internal state 0 by convention.

Transition Rules



In this situation:
I should move N.
I should enter state 2.

Picobot moves according to a set of rules:

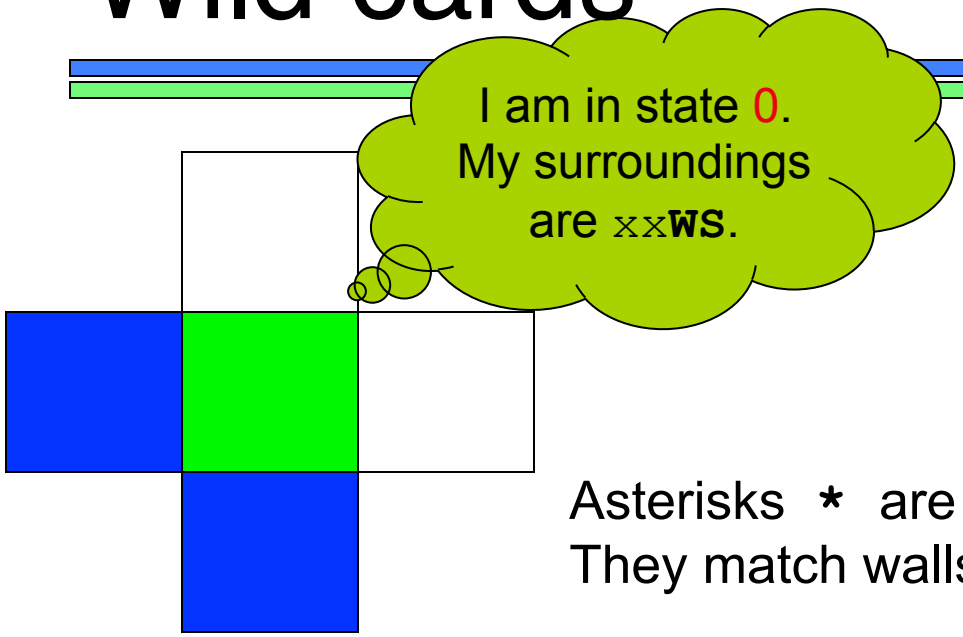
state	surroundings		direction	new state
0	xx**WS**	->	N	2

If I'm in state 0 *seeing* xx**WS**,

Then I move **N***orth, and change to state* 2.

Direction is one of {N, E, W, S, X}. Direction X means “no motion”.

Wild cards



Aha! This matches x***

Asterisks * are wild cards.
They match walls ***or*** empty space:

state	surroundings		direction	new state
0	x***	->	N	0

here, EWS may be wall or empty

Could we do without wild cards?

Global Consistency Requirement on a Picobot Rule Set

In any rule set, there can be **at most one** rule that applies to a given total state.

Thus the following **cannot occur together**:

```
2 *EWx -> N 3
2 x*** -> S 4
```

The **ordering** of the rules is not used to give priority to one rule over the other.
This can be both surprising and annoying, if you are used to thinking about checking rules in order.

Rule Consistency Error Messages

Rules

```
2 *EWx -> N 3
2 x*** -> S 4
```

Messages

```
Oops...

On line number 1, which is this:
   2 x*** -> S 4

Repeat Rule! The state: 2  surr: xEWx
   was already handled on line #0
   which reads as follows:
   2 *EWX -> N 3
```

Suggestion

When using the Picobot environment, keep a text editor handy on the side, so that you can record best attempts so far, etc.

Cut and paste using the mouse.

What will this set of rules do for Picobot?

state	surroundings		direction	new state
0	`x***`	->	N	0
0	`N***`	->	X	1
1	`***x`	->	S	1
1	`***S`	->	X	0

When Picobot finds a matching rule, that rule runs.

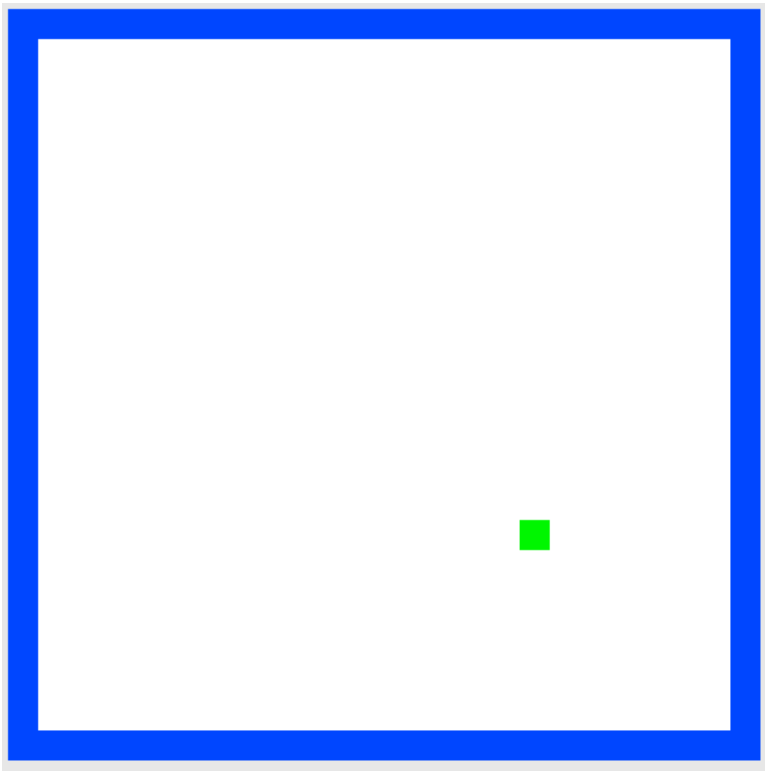
If no rule matches, Picobot will inform, and halt.

At most one rule is allowed per state and surroundings.

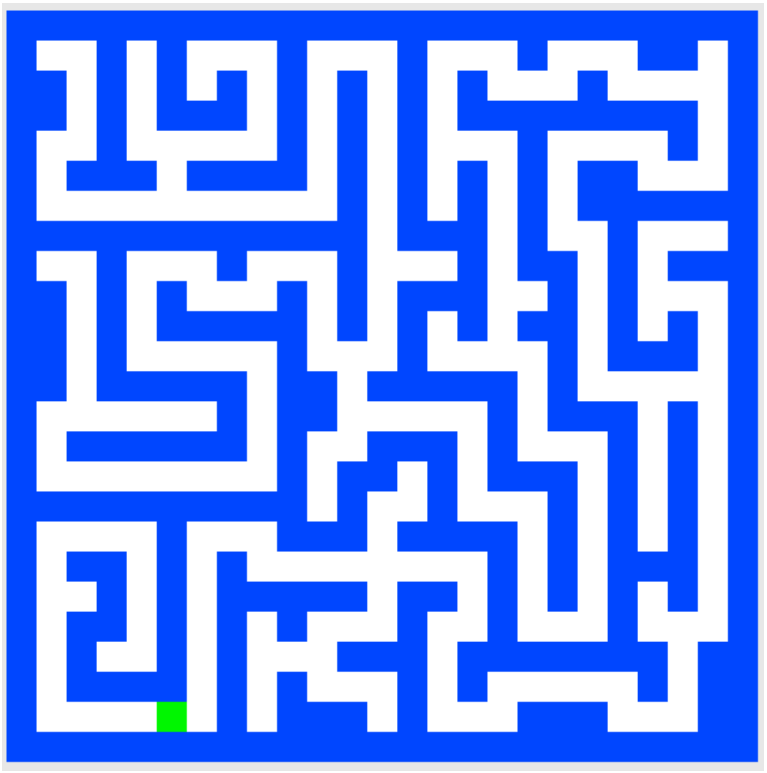
Problems

Construct rule sets that will always cause Picobot to completely cover these two rooms. Describe your strategies in English prose.

hw0, Problem #1



hw0, Problem #2



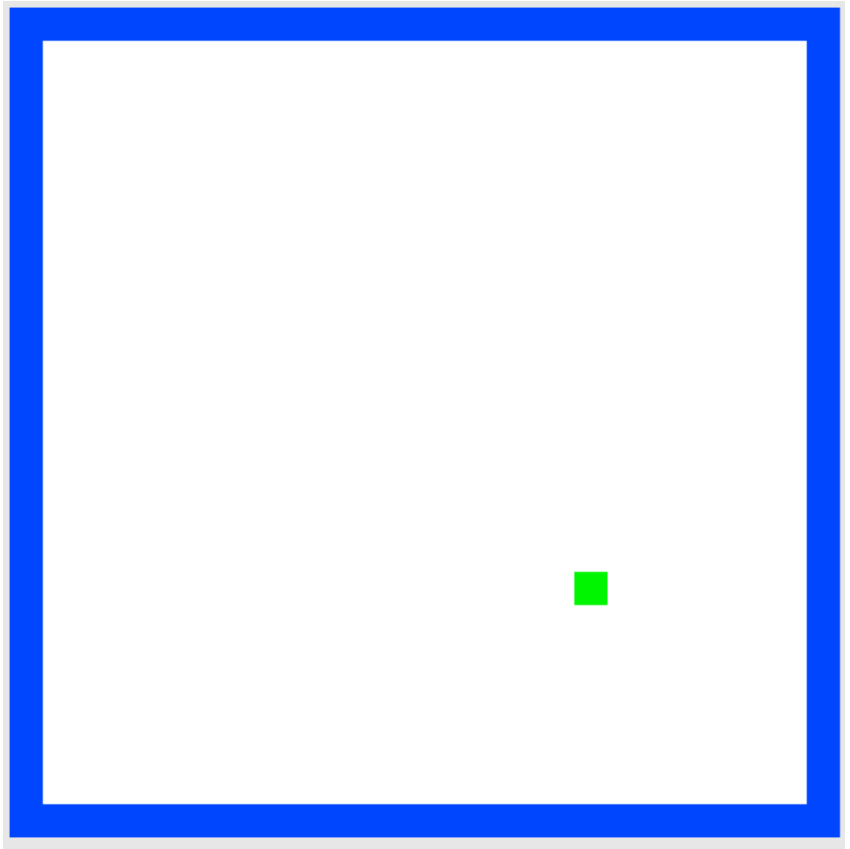
Each rule set must work *regardless* of Picobot's starting location.

“Quiz” #1

Alter these "up & down" rules so that Picobot will traverse the portion of the empty room to the left of the starting point.

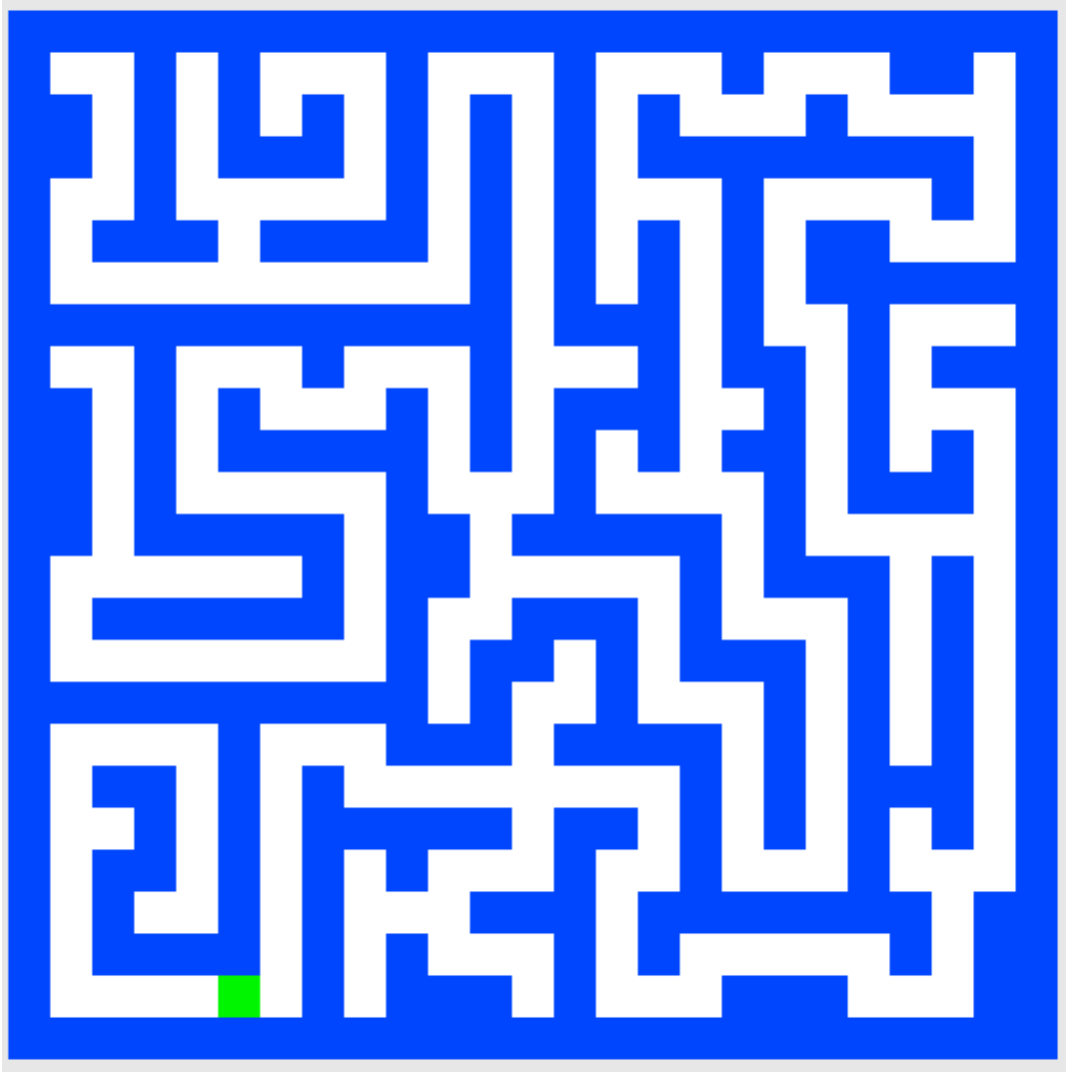
```
0    x***  ->    N    0
0    N***  ->    X    1

1    ***x  ->    S    1
1    ***S  ->    X    0
```



the empty room

Ideas for the maze?



the maze

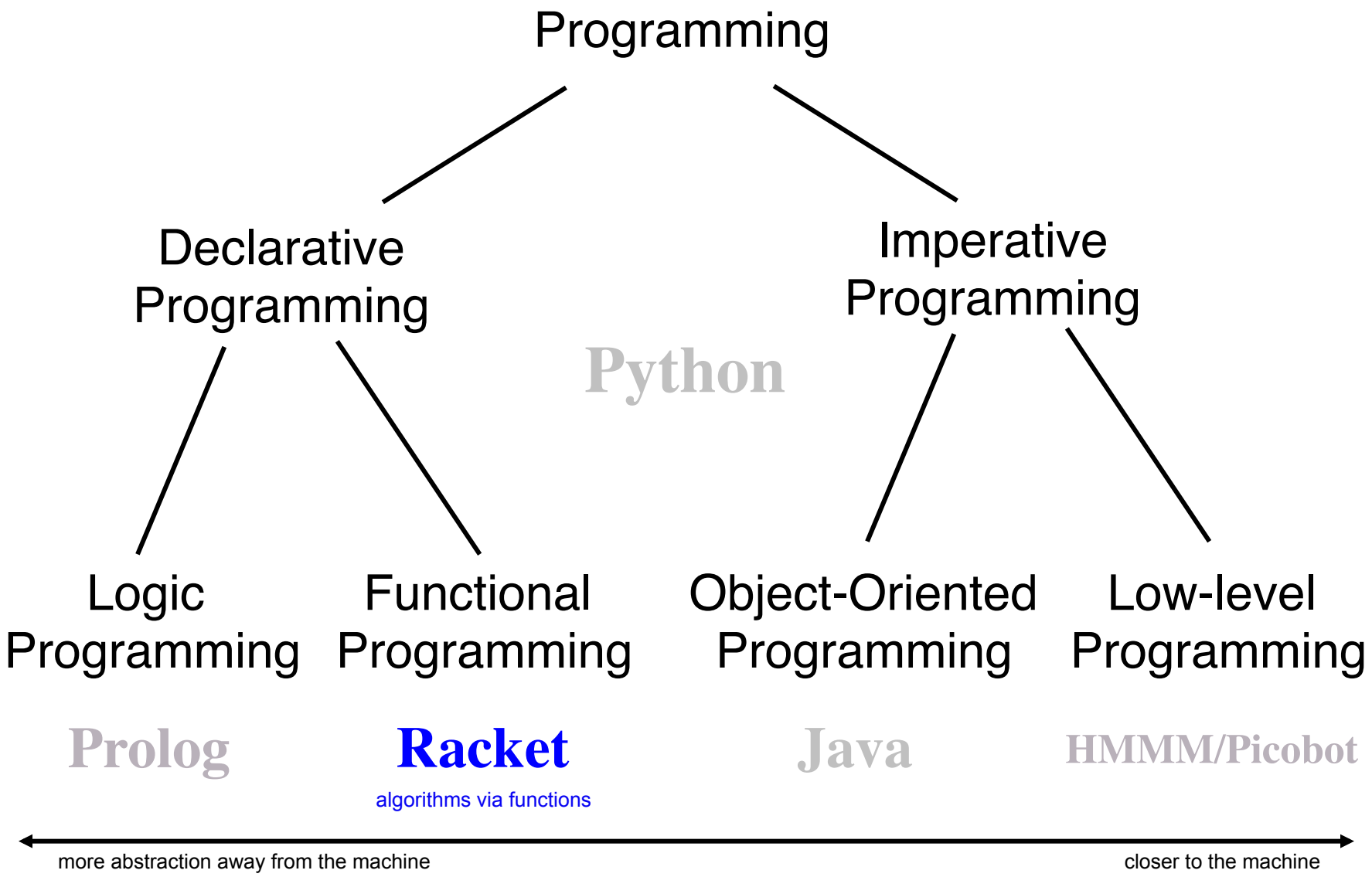
Extra Credit Problems

(may be hard)

- Is there a single rule set that will allow Picobot to cover *any* connected board (including, but not limited to, the examples given)? If so, exhibit such a rule set. If not, prove there is none.

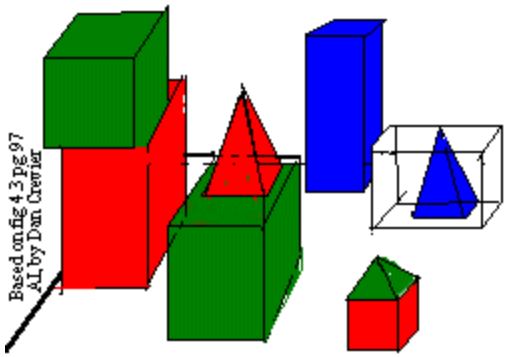
- Is there a 2-state rule set for the empty-room problem? Either show one, or prove there is none.

Onward to Programming Models



Racket

- Based on *Scheme*, which is based on LISP, one of the earliest AI programming languages



John McCarthy, 1927-2011,
Inventor of Lisp

http://en.wikipedia.org/wiki/John_McCarthy_(computer_scientist)

Why Racket / Scheme / Lisp?

#1 Elegance

#2 Minimal Syntax

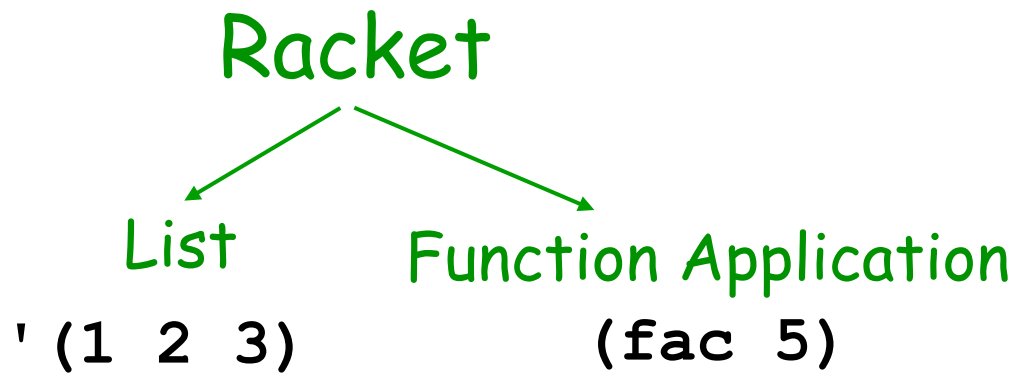
#3 Mathematical

#4 Programs as Data

“If you ain't got elegance
You can never ever carry it off.”
from “Hello, Dolly!”

Elegance

- *Functions* as primary building blocks — composability
- Lists are the main data structure — simplicity



- In fact, lists are ***everything*** - data and program

Ok, What is a Function, Anyway?

We are speaking of ***function*** in the *mathematical* sense.

Possible Definitions

A **function** is a rule such that, for any given value, called the *argument*, another value, called the *result*, is uniquely determined.

The emphasis here is on the word *uniquely*: A given argument does not determine more than one value.

An argument must determine exactly *one* value.

x: the argument f(x): the result

but in Racket, the result will be shown as (f x).

A Function can be depicted as a Table

Argument	Result
1	“Sunday”
2	“Monday”
3	“Tuesday”
. . .	. . .
28	“Sunday”
29	“Monday”
30	“Tuesday”

Sometimes the table would need to be **infinite** in order to show all results.

Functions as Sets of Pairs

A function can be regarded as a *set* of (argument, result) pairs.

Sometimes this set is called the ***graph*** of the function.

{(1, “Sunday”), (2, “Monday”), (3, “Tuesday”), … (30, “Tuesday”)}

The ***domain*** of the function is the set of first components of the pairs:
{1, 2, 3, …, 30}

The ***range*** of the function is the set of second components of the pairs:
{“Sunday”, “Monday”, “Tuesday”, …}

Which of these sets is a function?

{(0, “red”), (1, “green”), (2, “blue”), (3, “red”)}

{(0, “red”), (1, “green”), (2, “blue”), (0, “red”)}

{(0, “red”), (1, “green”), (0, “blue”), (2, “green”)}

{(0, “red”), (1, “red”), (2, “red”)}

{ }

Arity

Some functions have multiple argument value.
We can think of the arguments as being bundled as an n-tuple, and n is called the “arity” of the function.

Below we have a 2-ary function known as xor.

{((0, 0), 0), ((0, 1), 1), ((1, 0), 1), ((1, 1), 0)}

Constant Functions

A constant function is one in which

the result value is always the same.

A 0-ary function is always a constant function.

$$\{((), 99\}$$

is the graph of a constant 0-ary function.

A Function might be Computable

Example:

Given an argument,
the result is obtained by concatenating “-ary”:

This infinite function is exemplified by this table.

Argument	Result
“1”	“1-ary”
“2”	“2-ary”
“3”	“3-ary”
. . .	. . .
“googol”	“googol-ary”

What Functions *Don't* Do

Functions don't necessarily "do" anything.

Although we often use functions to describe the results of some computation or process, it is the process, not the function itself, that is the doing.

So technically, functions don't "take" arguments.

Also, functions don't modify their arguments.

Overloading

It is common in computer science, and most other fields, to find “overloaded” terminology.

This means that one word is used to mean more than one thing.

“function” is sometimes used to mean “procedure”, for example, where the arguments could be modified, the result is not unique, etc.

It is best to avoid overloading, but sometimes this is impossible due to historical precedent.

Side Effects

A procedure might not modify its arguments, and still might not qualify as a function.

For example, it could modify something else, such as a global value.

When a procedure does this, it is said to have a “side effect”.

Functions are representable by procedures that don’t have side effects.

Advantages of Not Having Side-Effects

(Discuss)

Functional Language

A programming language in which procedures have no side effects sometimes called a

"functional language".

This does not simply mean that the language "works",

just like "operating system" doesn't mean that the system "operates".

Mostly-Functional Languages

Many languages have a functional subset that is able to compute any computable function.

Some have non-functional components (ones with side-effects), that integrate with the functional ones.

Racket/Scheme is an example.

Racket/Scheme Notation

You are used to notation for functions results such as

f(x),
g(x, y),
…

Racket avoids commas and puts the function as the first element in an “S expression”:

(f x)
(g x y)

“S” stands for symbolic.

Racket/Scheme Terminology

Racket uses “procedure” to refer to both functions and procedures with side-effects.

Racket/Scheme Notation

Furthermore, this notation is used for what might have been infix notation:

2 + 3	becomes	(+ 2 3)
2 < 3	becomes	(< 2 3)

Just the Right Number of Parentheses

In Racket, parentheses are neither optional nor redundant:

(+ 2 3)

Not + 2 3

Not ((+ 2 3))

Not (+ (2) (3))

Some Functions have Arbitrary Arity

(+ 2 3)	2-ary
(+ 2 3 4)	ok, 3-ary
(+ 2 3 4 5 6)	ok, 5-ary
(+ 2)	ok, 1-ary
(+)	ok, 0-ary (result is the additive identity, 0)

This can be viewed as an example of overloading.

Primitive Data Types in Racket

Type	Meaning	Examples
Boolean	true or false value	#f, #t, (anything else behaves as if #t)
Integer	integer (arbitrary precision)	42, 12345
Rational	rational number	2/3
Float	floating-point numeral	2.34e-5
Complex	complex number	2+3i
Symbol	a unique thing in memory, with a printable value	foo
String	a string of characters	“this is one”
Character	a single character	#\c

List Data vs. Function Application

```
'(f 10)
```

Is a **literal list** consisting of two elements… ~ a "quoted" list

```
(f 10)
```

Is the value of the expression resulting when the function f is run on the single input, 10.

Uniform Syntax for Function Apps

A function application **always** appears as if a parenthesized list with:

The function first.

Zero or more arguments following

`(f 5)`	***1 argument***
`(g 3 4 5)`	***3 arguments***
`(h)`	***No arguments***
`()`	***Disallowed***

Special Forms

Some expressions resemble function apps, but really aren't. These are called **special forms** and are distinguished by their first element, which is a keyword.

```
(define (add5 x) (+ x 5))
```

The word ***define*** identifies this as a special form. This is **not** a function called "define" being applied to two arguments. Rather, it signals the definition of a function called ***add5***, which adds 5 to its argument.

Quoting is a Special Form

The ubiquitous **quote**, as in

```
'(to be or not to be)
```

is really an abbreviated special form:

```
(quote (to be or not to be))
```

Some Built-in Functions

`>`	arguments are numbers, value is boolean
`and or not`	boolean operators
`+ - * /` `modulo quotient`	arithmetic
`max min expt` `sqrt sin cos …`	mathematical functions

Conditional Special Forms

```
(if b
    true-branch
    false-branch)
```

A form that behaves like a function, almost.

```
(cond
   [ test     result ]
   [ test     result ]
   [ else     result ]
  )
```

Brackets are an alternative, to make it more readable.

Regular parentheses can be used instead.

Evaluation

"Evaluation" means the process of computing the result of an expression, such as, but not limited to, a function application.

This result is called the *value* of the expression.

We will show evaluations by a snapshot of an interaction with the Dr. Racket user interface.

```
> (+ 2 3)
5
```

Expression
Expression's value

More Evaluation Examples

```
> (+ 2 3)
5

> (+ 2 3 4 5 6)
20

> (+ 2)
2

> (+)
0

> (- 3 2)
1

> (- 3)
-3

> (-)
-: arity mismatch;
 the expected number of arguments does not match the given number
  expected: at least 1
  given: 0
```

More^2 Evaluation Examples

```
> 5                                          numbers self-evaluate
5

> (+ 2 (* 3 4))
14

> (* 7 (+ 2 4))
42

> (> 7 (+ 2 4))
#t

> (if (> 7 (+ 2 4)) 5 6)
5

> (if (= 7 (+ 2 4)) 5 6)
6

> (exp (* -i pi))
-1.0-1.224646799147353e-16i                  numerical roundoff error
```

Evaluation Process

If the expression is self-evaluating: return its value.

If the expression is a function application:
evaluate the function and each argument using this process. Then apply the function to the argument values.

Let **[** . . . **]** mean the value of expression . . . .

Evaluation Process

[5 **]** = 5 self-evaluation

[(+ 2 5) **]** = apply **[**+**]** to **[**2**]** **[**5**]**
= apply #<procedure:+> to 2 5
= 7

[(* 6 (+ 2 5)) **]** = apply **[***] to **[**6**]** **[**(+ 2 5)**]**
= apply #<procedure:*> to 6 **[**(+ 2 5)**]**
. . . what goes in here? …
= apply #<procedure:*> to 6 7
= 42

Evaluation Process for **if**

For if, we deviate from the basic process:

[(**if** B C D) **]**:

Compute **[** B **]**. If the result is #f, return the value of **[** D **]**. Otherwise return the value of **[** C **]**.

In other words, anything other than #f is interpreted as true.

What about the process for **cond**?

Names for the Evaluation Process

Racket uses **applicative order** evaluation.

The arguments are evaluated first, then the
the function is applied.

In some languages, the function is “applied” *before*
the arguments are evaluated. This is called
normal order evaluation. It is much rarer.

Special forms are evaluated in special ways,
which may be similar to normal order.