University of California, Berkeley – College of Engineering

Department of Electrical Engineering and Computer Sciences

Fall 2010 Instructors: Dan Garcia and Brian Harvey 2010-12-13

CS10 Paper Final Exam

Last Name	
First Name	
Student ID Number	
`cs10-` *Login First Letter*	`a b c d e f g h i j k l m`
`cs10-` *Login Last Letter*	`a b c d e f g h i j k l m` `n o p q r s t u v w x y z`
The name of your LAB TA (please circle)	**Jon** **Luke**
Name of the person to your Left	
Name of the person to your Right	
All my work is my own. I had no prior knowledge of the exam contents nor will I share the contents with others in CS10 who have not taken it yet. ***(please sign)***	

Instructions

- **Question 0 (1 point) involves filling in the front of this page and putting your login on the top of every sheet of paper.**
- This booklet contains 3 pages including this cover page. Put all answers on these pages; don't hand in any stray pieces of paper.
- Please turn off all pagers, cell phones and beepers. Remove all hats and headphones.
- You have 180 minutes to complete this exam. This final is closed book, no computers, no PDAs, no cell phones, no calculators, but you are allowed three double-sided pages of notes. There may be partial credit for incomplete answers; write as much of the solution as you can. When we provide a blank, please fit your answer within the space provided.

Question	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**Online**	**Total**
Points	**1**	**3**	**4**	**3**	**3**	**3**	**3**	**4**	**6**	**7**	**8**	**9**	**11**	**15**	**80**
Score															

Short-answer Questions (this page only)

Question 1 : We're told simulation is the third pillar of science, but these simulations could be based on bogus models of the world! How do scientists verify the *correctness* of their simulations? For example, how do they verify their models of climate change?

Question 2: With regard to HCI, we were in the mainframe era, then the PC era, and some declare that we are now entering a third era. What characterizes this new era (makes it distinct from the previous one), and what is an application / service / system that characterizes it?

Question 3: What are the technical requirements for (a) the mapper and (b) the reducer that allow MapReduce to work so effectively on thousands of machines at once?

Question 4: Cloud computing seems to be the panacea for data- or compute-intensive problems facing companies today, with very small upfront costs and seemingly infinite capacity on demand. With so much going for it, why are some companies staying away?

Question 5: What is the fundamental difference between the AI approach taken by chess programs (e.g., IBM's Deep Blue) and the AI approach in Dan's GamesCrafters research group for games much smaller than chess, such as Connect 4?

Question 6: Name three challenges in artificial intelligence that were once thought to be very difficult but are now in widespread use.

Question 7: (a) Name a problem that can't ever be solved by a computer, even in principle.

(b) A problem is considered intractable (in practice) if its order of growth is ______________.

Question 8: Are the authors of *Blown to Bits* optimistic or pessimistic about the future social implications of computing? Justify your answer with two specific examples.

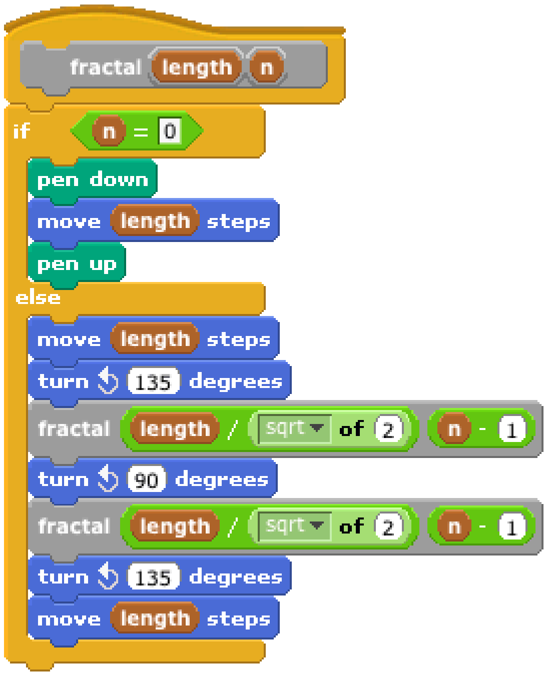
Login: **cs10-**____

Question 9: *Enter the (cousin of the) Dragon*

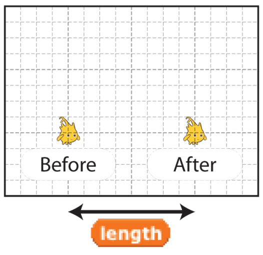
We've designed a fractal pattern (implemented by the program below) whose base case (n=0) simply draws a line *length* units long (leaving the sprite at the end of the line pointed in the same direction: that's important). The recursive case moves (with pen up) to the ending point, draws two smaller one-recursion-level-down copies to the "left" (where "left" is defined relative to the current orientation of the sprite) tilted at 45 degree angles (i.e., the legs of the isosceles right triangle whose hypotenuse is the length it just moved) ending up at the starting point, and then moves back to the ending point. (Reading the code will help you understand this.) The "L" in the drawing helps us remember which direction "left" is (i.e., which direction the next generation will bulge toward). The arrowheads aren't part of the drawing (and you don't have to draw them – we just drew them to remind you which direction "left" is). If you *do* draw the arrowheads, make them small.

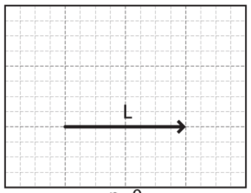
a) Draw the n = 2, 3, 4, and ∞ generations for the fractal. For each complete picture, we always start a drawing by lifting the pen (if it was down), clearing the screen, and moving the sprite to the "Before" point facing to the right. After the fractal is done, the sprite is positioned at the "After" point exactly **length** units away, as shown in the diagram below.

b) How does the number of line segments drawn scale with n? (i.e., what's its order of growth?) ____________________

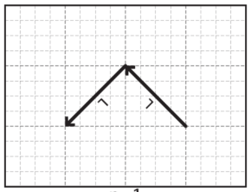


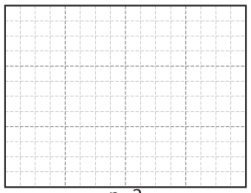
```
fractal (length) (n)
if <(n) = 0>
    pen down
    move (length) steps
    pen up
else
    move (length) steps
    turn ↺ 135 degrees
    fractal ((length) / (sqrt of 2)) ((n) - 1)
    turn ↺ 90 degrees
    fractal ((length) / (sqrt of 2)) ((n) - 1)
    turn ↺ 135 degrees
    move (length) steps
```

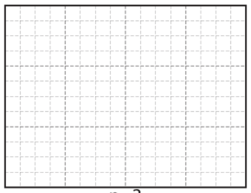


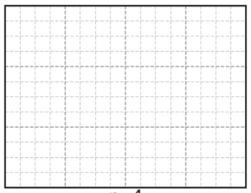


n=0

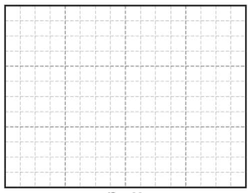

n=1


n=2


n=3


n=4

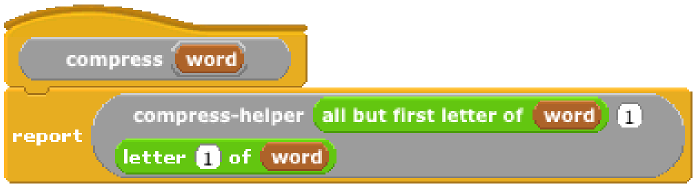
○○○


n=∞

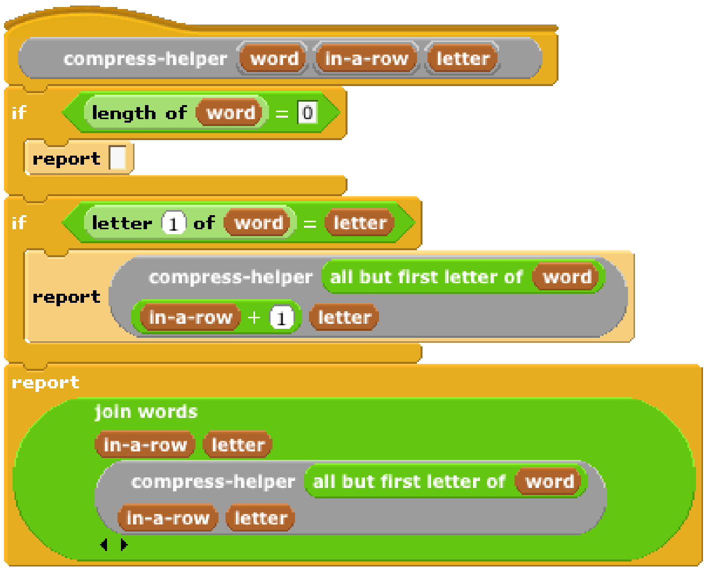
Question 10: *What I need now is a cold* *`compress`* *on my head!*

One way to save space with digital data is to *compress* it. You decide to write a block `compress word` to replace all consecutive characters of a word (including just single letters) with the consecutive number of them and the letter that's repeated. If the input were the exclamation "**`goooo!!`**" the output would be "**`1 g 4 o 2 !`**" (one "**`g`**", four "**`o`**"s, and two "**`!`**"s).

You try to write `compress word`, but it has two bugs (you'll need to find and fix them).



```
compress word
report compress-helper (all but first letter of word) 1
                       (letter 1 of word)
```



```
compress-helper word in-a-row letter
if (length of word) = 0
    report [ ]
if (letter 1 of word) = letter
    report compress-helper (all but first letter of word)
                           (in-a-row + 1) letter
report join words
           in-a-row letter
           compress-helper (all but first letter of word)
                           in-a-row letter
```

a) Currently, there's a problem because `compress aaa` reports ` ` instead of `compress aaa` reports `3 a`.
Cross off a single buggy line above (and write in the correction) to fix this bug.

b) Let's say you make the fix in part (a) above. There is one remaining bug. Show the *shortest sequence that triggers the bug* (and list the buggy and correct return values).

(Fill in the three blanks below, including the argument to the call to **`compress`** *below)*

Currently, `compress [          ]` reports ______________ instead of ______________.

Then fix it by replacing a single line as you did above (but this time cross it off with a squiggly line). *After fixing both bugs in (a) and (b),* `compress word` *should work for all valid input.*

c) Now, assume you've completely debugged your code above. If `length of word` reports `100`,

what is the maximum value of `length of (compress word)`? __________

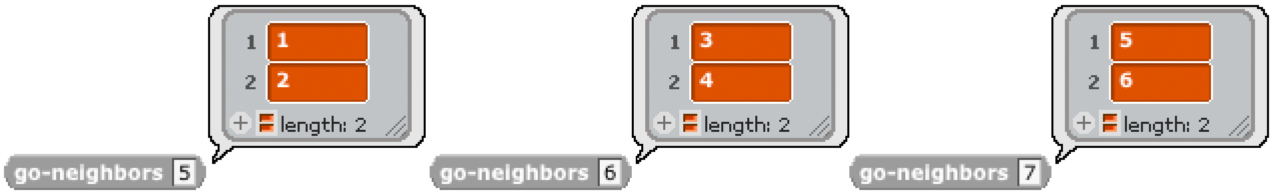
Login: **cs10-____**

Question 11: *Two^H^H^HMany roads diverged in a wood...*

We're sure you fondly remember the path-home? place problem from the midterm. We've reprinted it (with the answer) on the supplementary handout in case you've forgotten about it.

In computer science, one of the things we like to do is take *specific* problems and *generalize* them. What was specific in path-home? place? Each place had exactly *two* paths out of it (if it wasn't a dead-end), but that's not realistic. Most places have *many* paths out of them.

Fortunately, the problem statement won't change much. All we do is replace go-left place and go-right place with go-neighbors place which returns a *list* of neighbor places, each reached by going down a particular path out of that place. E.g., using the graph from the midterm version:



...but you could imagine some forests might have only one path out of a place, or a hundred! As before, calling go-neighbors place is an error if place is a *dead-end* , so go-neighbors place should never return a list with no elements.

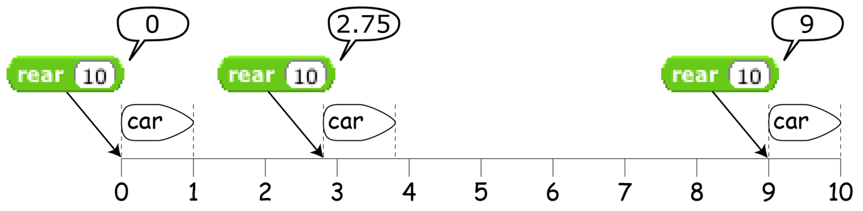
Change what is needed from the midterm answer (originally in BYOB but translated below to paper) to account for the change. Cross off code to delete it, and/or insert code where appropriate.

```
path-home?(PLACE)
 if(home?(PLACE))
   report(true)
 if(dead-end?(PLACE))
   report(false)
 report( path-home?(go-left(PLACE)) or path-home?(go-right(PLACE)) )
```

Question 12: *Finding Parking in Berkeley. Get it?*

Have you ever tried to park on a side street in a Berkeley residential area? People do such a poor job parking that they waste most of the street. How many 1-unit-long cars can park on a street given that people park randomly? Let's simulate it! We're going to assume cars don't need any extra "breathing room" between them to park and can just "drop" into a tight space.

Luckily, someone else has provide a helper block called `rear (space)`, which takes in the amount of space left, and *randomly* picks a place for the car to park in that space, reporting *the location of the car's rear end* (hence the name). It is an error to call `rear (space)` with less than 1 unit of free space available. As an example, a call to `rear (10)` would return a number from 0 (car parked in the *back* of the 10-unit spot) through 9 (car parked at the *front* of the 10-unit spot) or any number in-between, like 2.75. All three example reported values are shown below.



a. In the *best* case, we can get 10 cars to park in a 10-unit space.
How many 1-unit-long cars can park in a 10-unit space in the *worst* case? _____

b. Fill in the blanks to complete a block called `number parked (space)`, which takes in the amount of space in the street, and simulates random parking to estimate the number of cars that are able to park on the street before there's no more space. *Hint: Think of what happens when the first car randomly parks ... it creates two new spaces: front & behind.*

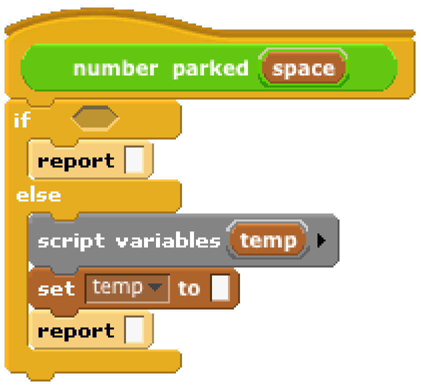
```
number-parked(SPACE)

    if ( ____________________________ )

        report ( __________________________ )
    else
        script-variable(TEMP)

        set-(TEMP)-to-( ______________________ )

        report ( ____________________________________________________________ )
```



```
number parked (space)
if < >
    report [ ]
else
    script variables (temp)
    set [temp] to [ ]
    report [ ]
```

Example: `number parked (space)`

Midterm “`path-home?`” Question with Answer

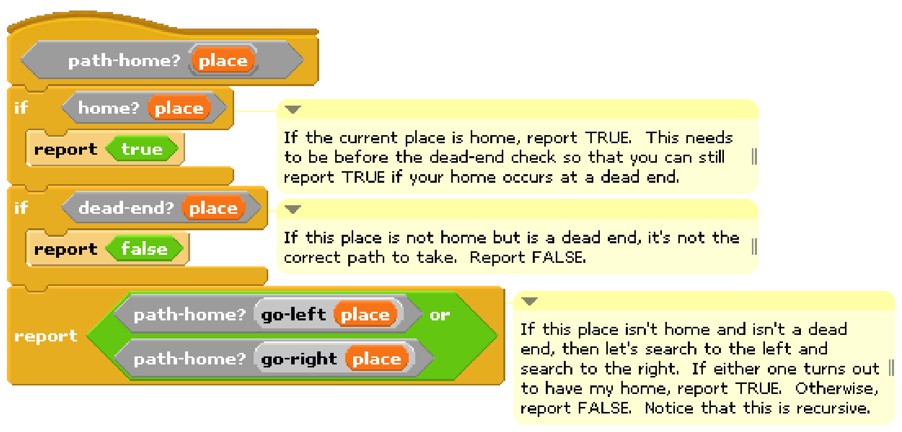
You’re lost in the forest. Every place in the forest is either a *dead-end* or has exactly 2 *one-way paths*: *left* and *right*. Your goal is to find out if there is a way home. We introduce a new data type called a place, but you don’t know *(and you don’t need to know)* how it is represented; it could be a string, a number, or a list. You are presented with four new blocks, two predicates and two reporter blocks (all take a place as an argument):

- home? place returns true if the place is your home, false otherwise.
- dead-end? place returns true is the place is a dead-end (i.e., no paths from it).
- go-left place follows the *left* path, returning a new place.
- go-right place follows the *right* path, returning a new place.

It is an error to go-left place or go-right place if place is a *dead-end* (because it has no paths!). There is no way in this forest to follow a sequence of left paths and/or right paths and end up where you started. I.e., there’s no way to walk in circles. Your *home* (if one exists) might be at a dead-end or it might not. You might actually start your search at home.

Write path-home? place, which uses the four functions above and returns true if you can get home following a (possibly zero) number of lefts and rights starting from place, and false otherwise. Use the technique we described for authoring BYOB code on paper. We’ve provided an example forest for you, but **your solution needs to be able to work with ANY forest.** Below, we present a table that shows the responses of various blocks when you are at different places in the sample forest on the lower right.

place	home? place	dead-end? place	go-left place	go-right place	path-home? place
1	false	true	ERROR	ERROR	false
2	false	true	ERROR	ERROR	false
3	false	true	ERROR	ERROR	false
4	false	true	ERROR	ERROR	false
5	false	false	1	2	false
6	true	false	3	4	true
7	false	false	5	6	true

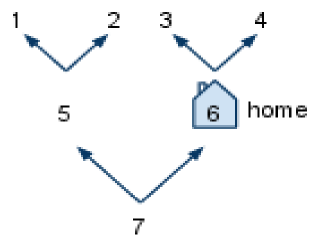


```
path-home? place
if home? place
    report true
if dead-end? place
    report false
report (path-home? go-left place) or (path-home? go-right place)
```

If the current place is home, report TRUE. This needs to be before the dead-end check so that you can still report TRUE if your home occurs at a dead end.

If this place is not home but is a dead end, it's not the correct path to take. Report FALSE.

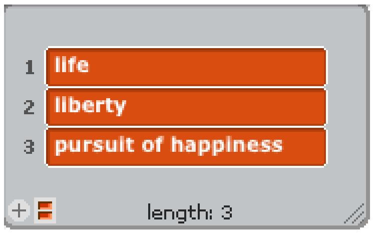
If this place isn't home and isn't a dead end, then let's search to the left and search to the right. If either one turns out to have my home, report TRUE. Otherwise, report FALSE. Notice that this is recursive.



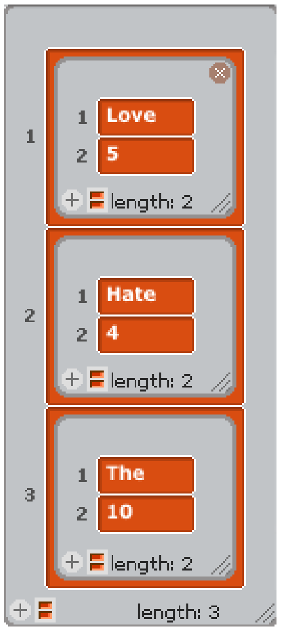
Writing Scratch/BYOB code on paper

You might be asked to write Scratch/BYOB code on exams, so we've developed a technique for writing it on paper. There are a few key things to notice:

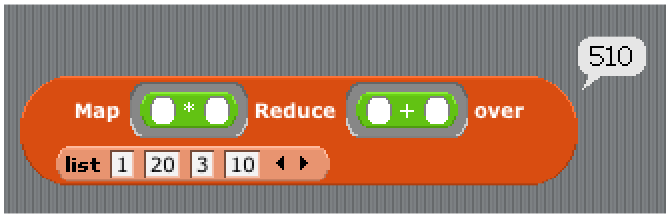
- We write variables in **`UPPERCASE`**.
- We change spaces between words in block names to dashes (this makes it much easier to read).
- Parentheses mark the start and end of a parameter list, and we separate consecutive parameters by commas
- We use indentation just as Scratch/BYOB does, to help us understand what is "inside" the **`if`**, **`else`**, and other Control structures.
- When you want to write a list of things, write them with an open parenthesis, then the first item, second item, etc (separated by spaces) and when you're done, put a closed parenthesis. If any of your items are a sentence, you have to put quotes around the sentence. So, for example, the following list of three things would be written as the equivalent 3-element-list:
  - **`(life liberty "pursuit of happiness")`**.



- Similarly, a nested list just shows up as a nested set of parenthesis. So the following would be written as
  - **`((Love 5) (Hate 4) (The 10))`**.

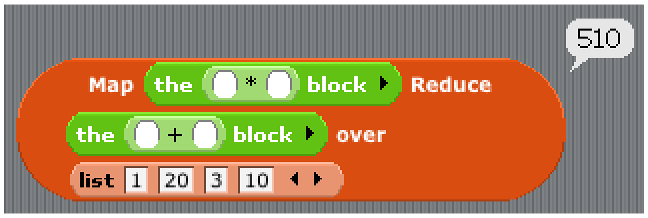


- If you want to pass in a function as argument, you have two options in BYOB: use the grey-border or the more verbose **`the( )block`** green block. Here are three new conventions:
  - The grey border is written as square brackets: **`[ ]`**
  - Blanks are written as parenthesis with underscore _ in the middle, but common blocks that are passed in to HOFs can be simplified by just their name (and not the parens and underscores)
  - Return values are written as **`==> value`**
- So the following would be written as:
  - **`Map[ (_)*(_) ]Reduce[ (_)+(_) ]over( (1 20 3 10) ) ==> 510`**
- or, in the more simplified (and preferred) format:
  - **`Map[ * ]Reduce[ + ]over( (1 20 3 10) ) ==> 510`**

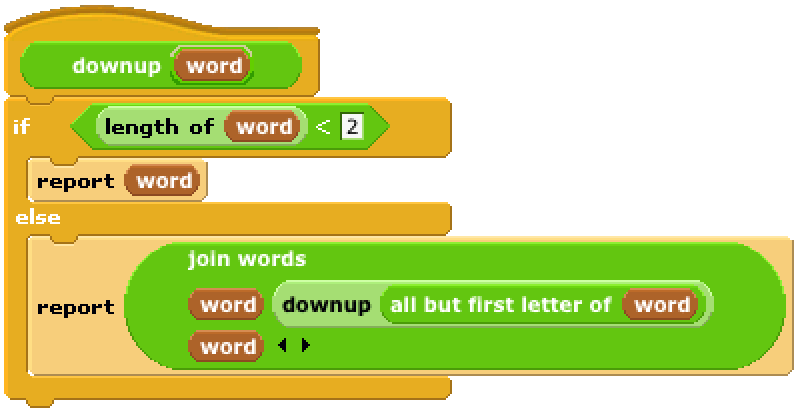


- If you prefer to use the **`the( )block`** green block, it could also be written:

        - `Map(the( (_)*(_) )block)Reduce(the( (_)+(_) )block)over( (1 20 3 10) ) ==> 510`
    - or, in the more simplified (again, preferred) format:
        - `Map(the(*)block)Reduce(the(+)block)over( (1 20 3 10) ) ==> 510`



Here's a sample (and a familiar piece of BYOB code):



...and here's how we would write it on an exam using our technique:

```
downup(WORD)
   if length-of(WORD) < 2
      report(WORD)
   else
      report(join-words(WORD, downup(all-but-first-letter-of(WORD)), WORD))
```

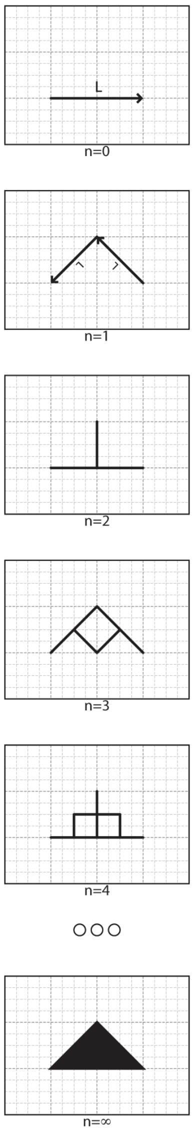
Here's how you could write the **`factorial-of`** block from lab.

```
factorial-of(NUM)
   if NUM = 1
      report(1)
   else
      report(NUM * factorial-of(NUM - 1))
```

2010Fa CS10 Paper Final Answers

Question 1: Compare with empirical data. In the case of climate simulation, that means comparing against weather data from the past and seeing if the predicted weather is what actually happened (e.g., the 1938 hurricane that hit New York).

Question 2: It's an era of mobile, social and ubiquitous computing. Examples? Tweeting / facebook from a smartphone, exploring a city via foursquare, using Urbanspoon or Yelp to choose a restaurant, etc.



Question 3: They both must be functions, i.e., whose value is dependent only on the input and not on some prior state. In addition, the reducer should be associative and commutative. For the simplified model that we demonstrated in class, the domain of the reducer must include the range of the mapper AND the range of the reducer (not required for full credit).

Question 4: "Concern about..."

1. service availability (trusting your business to someone else)
2. data lock-in (getting your data out when needed)
3. security / confidentiality / auditability
4. data transfer bottlenecks
5. performance unpredictability
6. scalable storage (as easy as computation)
7. bugs in large-scale distributed systems
8. the ability to scale up quickly
9. reputation fate sharing. E.g., spam-prevention, unexpected downtime for subpoenas, etc
10. software licensing (current non-open-source software pricing models don't scale well)

Question 5: Since chess can't be solved, chess AIs make moves based on "best guess" evaluations of the board. Dan's research group focuses on smaller games (and puzzles) that can be *strongly solved*, so his system *knows* the answer, once and for all, and plays perfectly.

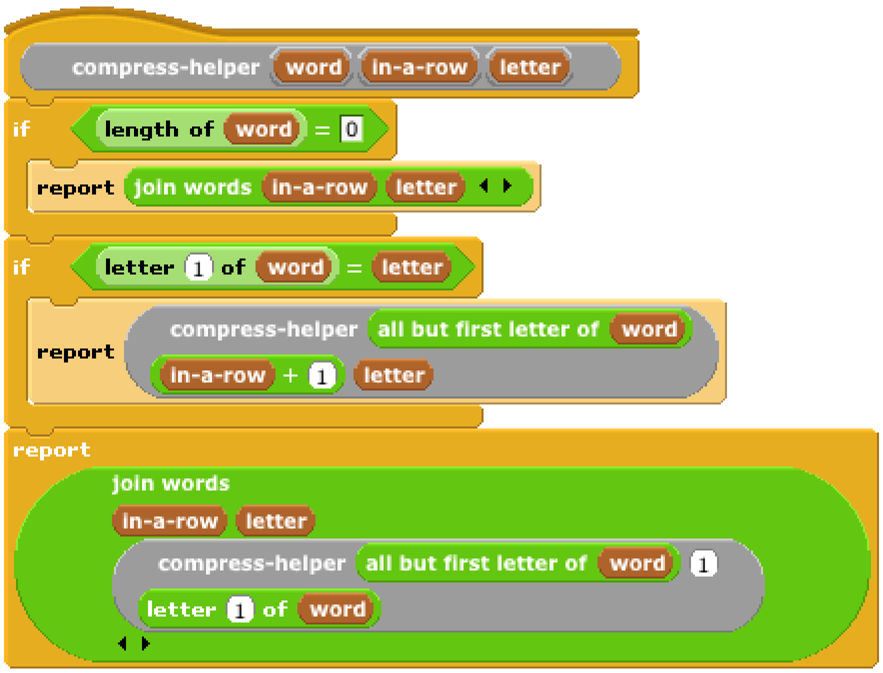
Question 6: Playing chess, voice recognition, natural language translation, gesture recognition, driving trains, etc.

Question 7: (a) Halting problem. (b) exponential.

Question 8: My answer is "optimistic," although a case could be made for either. Some examples: They say in the last chapter that they think the Internet will eventually penetrate even currently isolated populations such as North Korea. They think that decreasing regulation will solve technological problems. They predict that people will come not to mind the loss of their privacy (although they have mixed feelings about that themselves).

Question 9: (a) The n = 2, 3, 4, and ∞ generations for the fractal are above. (b) Exponential.

Question 10: (a) The first reported value changes to "`join-words(in-a-row, letter)`"
(b) `compress("hi")` returns "`1 h 1 h`" instead of "`1 h 1 i`"; the fix involves changing the last reported value to `compress-helper(all-but-first-letter-of(word), 1, letter(1)of(word))`
(c) 399. Alternating letters (e.g., "`hihi`...") double in size with spaces "`1 h 1 i 1 h 1 i`..."



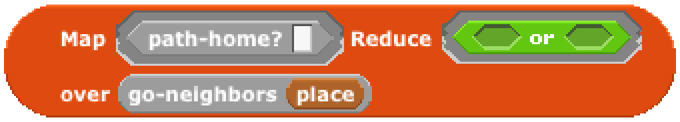
```
compress-helper word in-a-row letter
if < (length of word) = 0 >
    report join words in-a-row letter
if < (letter 1 of word) = letter >
    report compress-helper (all but first letter of word) (in-a-row + 1) letter
report
    join words
    in-a-row letter
    compress-helper (all but first letter of word) 1
    (letter 1 of word)
```

Question 11: Change the value returned in the `report` block on line 6 to (4 options, depending on whether they used our MapReduce or not, and grey borders or `the( )block` green block):

`Map[ path-home? ]Reduce[ or ]over( go-neighbors(PLACE) )`

or

`Map[ path-home?(_) ]Reduce[ (_)or(_) ]over( go-neighbors(PLACE) )`

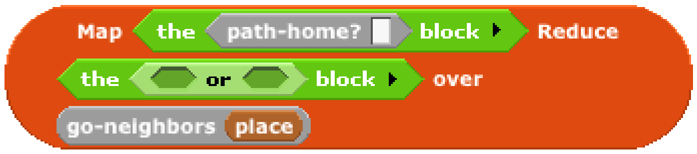


```
Map (path-home? ) Reduce (< > or < >)
over (go-neighbors place)
```

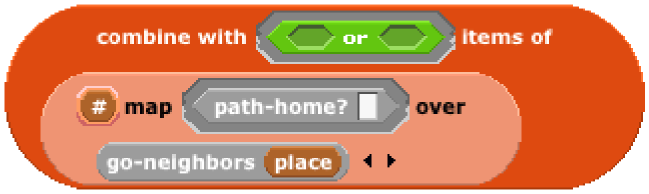
`Map(the(path-home?)block)Reduce(the(or)block)over( go-neighbors(PLACE) )`

or

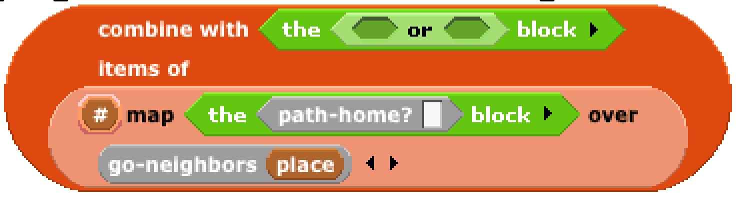
`Map(the(path-home?(_))block)Reduce(the((_)or(_))block)over( go-neighbors(PLACE) )`


```
Map (the (path-home? [ ]) block) Reduce
(the (< > or < >) block) over
(go-neighbors (place))
```

```
combine-with-[ or ]-items-of( map[ path-home? ]over( go-neighbors(PLACE) ) )
```
or
```
combine-with-[ (_)or(_) ]-items-of( map[ path-home?(_) ]over( go-neighbors(PLACE) ) )
```


```
combine with (< > or < >) items of
  (# map (path-home? [ ]) over
     (go-neighbors (place)) ◂ ▸)
```

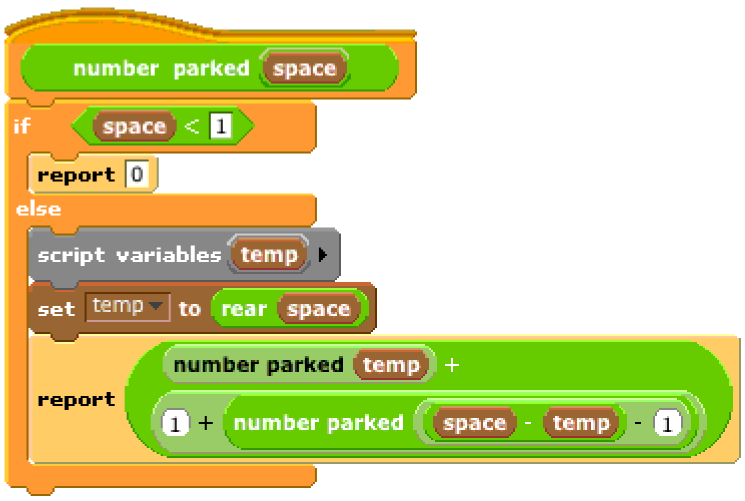
```
combine-with-(the(or)block)-items-of( map(the(path-home?)block)over( go-neighbors(PLACE) ) )
```
or
```
combine-with-(the((_)or(_))block)-items-of(map(the(path-home?(_))block)over(go-neighbors(PLACE)))
```


```
combine with (the (< > or < >) block)
items of
  (# map (the (path-home? [ ]) block) over
     (go-neighbors (place)) ◂ ▸)
```

Question 12:

a. 5. Assuming the biggest space that *cannot* hold a car is = 0.99, we place the cars from back to front so that we always leave a 0.99 spot between cars, the worst possible.
   1. 0.99 - 1.99
   2. 2.98 - 3.98
   3. 4.97 - 5.97
   4. 6.96 - 7.96
   5. 8.95 - 9.95

b. It's a simple divide-and-conquer recursion:


```
number parked (space)
if <(space) < [1]>
    report [0]
else
    script variables (temp) ▸
    set [temp] to (rear (space))
    report ((number parked (temp)) +
            ((1) + (number parked ((space) - (temp) - (1)))))
```