

Unit 1 Materials: Introduction to Computers and Programming

The first day of this project exposes students to the programming environment they'll be using for the entire project and starts them thinking about computers, programming, and the specific problem — search — that they'll work on for the next few weeks. By the end of this day, the students should know enough about the programming environment to be comfortable opening it, playing with it, and seeing what happens. They should have a sense of what they'll achieve over the course of the project: basic skill with a programming language and environment and an understanding of one of the most fundamental problems of computer science. Finally, they should understand the “API” we've layered on top of StarLogo well enough to use some of the functions to control a turtle and to phrase questions about other API calls.

Opener: What is a Computer? (And, what isn't?)

Hand out a series of cards to the groups, each group gets four or five cards (some overlap between groups is fine and even desirable). Each card has the name of a device and a brief description of the device on it. The groups are asked to look over the cards and decide, as a group, which of the devices are computers, which are not, and why. To make this more fun, you might consider giving each group at least one of the more amusing cards like “domesticated cat” or “human brain.” The groups' specific assignment is to create two piles of cards, one containing things that are computers and one containing things that aren't and to be able to justify that division.

When they're done, bring them back together and discuss people's categorizations and why they made their decisions. Try to come to consensus on a few interesting cards. The class should, of course, be free to construct a division according to their own collective understanding of what a computer is. However, it's important that an understanding of computers as *devices that follow instructions* comes out of this exercise. The moderator should ensure that the idea of following instructions enters the discussion before the end of the exercise. A second, less important note, might be to introduce some discussion about whether some of these listed items *are* computers or *contain* computers.

By the way, here's an excellent quote from *Computers, the Internet and Criminal Justice: Literacy and Beyond* by Cecil Greek describing the line between calculating devices and computers: “[Babbage's analytical] engine could choose to perform the next operation depending on how the previous operations turned out, and was not limited to sequential calculations producing static tables. Real computers allow symbolic manipulation, not just numeric operations.”

Some notes on interesting comparisons/contrasts below. The abacus, arithometer, and analytical engines are all early computing devices (with very different definitions of early). Of these, I believe the analytical engine is a computer. The ENIAC, Pentium II microprocessor, iMac, and Xbox are all things that are recognized today as computers (well, the PII is at least a part of a computer). I no longer believe the ENIAC is a true computer. It had no stored program capability and, more importantly, no conditional execution. The Xbox seems to have knocked off HAL's trademark “eye.” (Compare the two pictures on the cards!) The AmigoBot, marionette, toaster, and VCR are all “direct control” types of devices. It might be fun to pair one of the two more “computerish” ones with one of the two less computerish ones. I tend not to believe that any of these things are, in fact, computers. Finally, a fairly strong argument can be made to disqualify the VCR and AmigoBot and (much less so) the Xbox from being computers because they are not *general purpose* computing devices.

Cards:

- Abacus

Developed ca. 3000 B.C. in Asia Minor.

A device consisting of rows of beads mounted on rods. The beads can slide along the rods, and different positions of beads represent different numerical values. The operator of an abacus can follow certain well-defined procedures to perform calculations surprisingly quickly. In the mid-twentieth century, an abacus outperformed an electric calculator in exciting, head-to-head, calculating competition.

[Jones Telecommunications & Multimedia Encyclopedia:

http://www.digitalcentury.com/encyclo/update/comp_hd.html]

- Arithometer

Invented in 1820 by Charles Xavier Thomas de Colmar

A mechanical device (using cogs and gears) capable of adding, subtracting, multiplying, and dividing. This calculating device enjoyed wide popularity as late as World War I. Like a simple calculator, the user of the arithometer would command it to perform a calculation (by pushing buttons and moving levers), but the arithometer itself could not use the results of its calculations to decide what calculations to perform next.

[Jones Telecommunications & Multimedia Encyclopedia:

http://www.digitalcentury.com/encyclo/update/comp_hd.html]

- ENIAC

Developed in 1944 by Mauchly, Eckert, and other scientists at U. Penn.

An electronic device which used vacuum tubes, electrical relays, resistors, and other components to perform calculations. Operators would wire together various parts of the ENIAC's electrical devices to represent the set of commands it should execute. It could then execute those commands, performing literally thousands of additions in a single second. The ENIAC could not, however, decide what commands to execute next based on the results of its previous calculations. Physically the ENIAC was over 150 feet in width and took up a large room.

[UPenn Almanac article: <http://www.upenn.edu/almanac/v42/n18/eniac.html>]

- Analytical Engine

Designed (but never built) ca. 1832 by Charles Babbage in collaboration with Ada Lovelace (Augusta Ada King, Countess of Lovelace).

A machine which would have received instructions from punch-cards — pieces of paper with holes punched in them in patterns with meaning to the machine. The machine would perform the calculations described on the cards using steam power and a complex system of gears and cogs. As it proceeded, the results of previous calculations could be used to control what calculations it performed next. Finally, it would print its results.

[Jones Telecommunications & Multimedia Encyclopedia:

http://www.digitalcentury.com/encyclo/update/comp_hd.html]

- Pentium II Microprocessor

Developed ca. 1997 by Intel Corp.

A tiny, square wafer of silicon which has been printed — through “etch lithography” and “doping” processes — with millions of transistors. Each transistor is capable of blocking or passing a signal based on a second signal which controls it. The pattern printed on the Pentium II allows it to perform millions of computations per second based on commands supplied electrically from pins — small metal prongs which attach to the chip. It provides the results of these calculations as well as requesting new commands or data electrically through other pins. The Pentium II is capable of storing its commands electronically and choosing which commands to execute based on previous calculations. The Pentium II takes up a few square millimeters on a wafer-thin chip.

[Intel Microprocessor Hall of Fame: http://www.intel.com/intel/intelis/museum/exhibit/hist_micro/h

- iMac

Developed in 1997 by Apple Corp.

The iMac is a single system comprised of a monitor, hard drive, CD drive, memory, a PowerPC G3 processor, and the Macintosh (OS X) operating system. An iMac can receive commands from several of its own components, from the internet, or from a keyboard, mouse, or other control device. It can be used to play games, edit documents, and a variety of other tasks. It also comes in quite a few oddly named colors like “Snow,” “Indigo,” or “Graphite.”

[Apple: <http://www.apple.com/imac/g3/>]

- HAL9000

Developed ca. 1991 (sort of) by Arthur C. Clarke (sort of)

An advanced device capable of performing a variety of tasks and interacting with its human users (companions?). The HAL9000 communicates by voice and can control auxiliary devices such as the pod bay doors on a

spaceship. It (he?) has an unfortunate tendency towards obsessing over minor details or inconsistencies in the instructions given it, however. In the events described in Arthur C. Clarke's "2001: A Space Odyssey," HAL's tendency toward obsessive literalism led to the unfortunate death of most of its spaceship's human crew (oops).

- Microsoft Xbox

Developed in 2001 by Microsoft Corp.

The Xbox is a gaming system: it reads commands from a DVD and executes the commands along with responding to input from control devices to display games on an attached monitor. The Xbox can be connected to the internet through an Ethernet port, contains a hard drive to store information, and runs on a 733MHz Intel processor. The Xbox, however, has never controlled a mission to space.

[Microsoft: <http://www.microsoft.com/xbox/>]

- AmigoBot

Developed late 20th century by ActivMedia Robotics.

The AmigoBot is a small, wireless, mobile robot. It weighs about 10 pounds (with batteries) and can run for approximately 2 hours before charging. It is equipped with 10 sonar sensors (devices that send out a sound signal and then time how long it takes the signal to return) and sensors that measure the movement of its wheels. It can also be equipped with a "hand" to manipulate objects in the world. It receives commands directly from a joystick, wirelessly from a computer, or from a computer resting on top of the robot wired in directly.

[ActivMedia Robotics: <http://www.amigobot.com/amigo/techspecs.html>]

- Marionette

Popularized mid-19th century by Thomas Holden.

A marionette is a puppet controlled from above by a set of strings. A single, basic marionette has about nine strings, but more complex puppets can have dozens. Each string controls the position of some part of the puppet, and coordinated movement of the strings (by tilting a fixed framework) can elicit a variety of compelling motions. "Plucking" individual strings can create other complex movements. Marionettes range widely in size (see *Being John Malkovich*).

[Encyclopedia Britannica: <http://www.britannica.com/eb/article?query=marionette&eu=117364&tocid=>

- Timed Pop-up Toaster

Developed in 1919 by Charles Strite

Using a system of springs, heating element, and timer, the pop-up toaster can toast bread — caramelizing surface sugars to a golden-brown, crunchy consistency — without the need for human supervision.

[Toaster Museum: <http://www.toaster.org/museumintro.html>]

- VCR

First commercialized for home use by Sony in 1969.

A VCR (Video Cassette Recorder) is a device that transcribes video signals (such as television) onto a magnetic tape. The video signal can then be replayed from the tape. Some modern VCRs can quickly advance or rewind their tapes, leave marks on the tapes to which they can return, automatically scan past commercials in recorded television broadcasts, or automatically record television broadcasts at preset times and dates.

[Encyclopedia Britannica: <http://www.britannica.com/eb/article?eu=77264&tocid=0&query=vcr>]

- Domestic Cat

Domesticated ca. 2000 B.C. in Egypt.

A four-legged creature with a tail, covered in fur and possessed of retractable claws. Cats pretty much do whatever they want. Physically, the domestic cat weighs between about 5 and 20 pounds.

[Natural History Museum of LA County: <http://www.lam.mus.ca.us/cats/home.html>]

- The human brain

Year of development in dispute (ca. 100,000 - 200,000 B.C. according to “Out of Africa” evolutionary theory). Developer also in dispute.

The human brain is a large organ, one of which is situated within the skull of each modern human. Capable of a wide range of tasks from mathematical calculations to identifying Jelly Bean flavors, the human brain coordinates most functions of its associated human. The human brain receives sensory information from a variety of devices (eyes, ears, toes, etc.) and also controls a variety of devices capable of acting in the world (hands, voice box, toes, etc.). The structure of the human brain consists of a massive network of interconnected neurons, a specialized kind of cell, interacting through chemical and electric processes. The human brain receives power from edible objects in the world around it like Jelly Beans. Physically, the human brain weighs in at about three pounds.

- Microsoft Windows

Developed ca. 1985 by Microsoft Corp.

Software which regulates the control of a computing device by programs. Windows handles the operation of the machine’s physical devices, presents an interface for the user to the machine’s storage devices, supports other programs’ interfaces, and a wide variety of other functions.

What is Programming?

The instructor should briefly explain what programming is. She might instead hand out a sheet with information like the text below and have students read it for a minute or two and then discuss it. Here’s text that the instructor can use as a basis for this part:

We’ve decided a computer as (among other things) a device that follows instructions. In fact, computers very precisely follow instructions, doing exactly what the instructions tell them to do. To get that sort of precision, the instructions themselves need to be very clear and precise. Therefore, computer instructions are given in a “language” with a set of well-defined statements or commands. Unlike human languages (e.g., English), there are rules to computer languages that describe exactly what a statement in the language means.¹

We communicate instructions to computers with programs. A program is a sequence of instructions to the computer. If a computer uses the language a program is written in (the “programming language”), it can “execute” the program: doing exactly what the instructions in the program tell it to do. A program might contain instructions that cause your computer to draw a line across its screen or calculate the value of 23^2 . Microsoft Internet Explorer and Netscape Navigator are also programs (although much more complex!) with instructions that cause your computer to contact other computers, download, and display a web page when you click on a link.

How are these programs created? As a rule people write them. (It’s actually an incredibly difficult problem to have a computer write its own programs. Indeed, we can *prove* that it’s impossible in its most general sense! However, the proof is quite complex.) The act of writing a program and all the other acts we perform while writing a program (like planning how to write it, testing to see if it works, writing notes describing how it works, etc.) are called “programming.”

During this project you’re going to learn how to program a computer in a programming language called “StarLogo.” You’ll write precise instructions that the computer will follow to solve problems.

Programming and Computer Science

The instructor should explain how programming relates to computer science. Here’s some text that might help:

“Computer science and computer engineering” are fields that study computers and programming. Part of these fields is building computers and writing the programs that make them do useful work, but these fields also study how computers work, what computers can (and cannot) do, how to write programs, how people interact with computers, how computers relate to other fields, etc. Computer science might involve proving that computers aren’t able to perform a certain operation (like writing programs!), developing a new computing device, discovering an interesting problem that we can use computers to solve, or inventing ways to make computers more useful for people.

¹This might be a good point to discuss briefly whether there *are* rules that describe exactly what an English statement means.

Understanding what a computer is and how they perform tasks — by executing programs — is at the heart of all of these activities, however. Furthermore, computer scientists often write programs to help solve the problems they are interested in.

Our Problem

The instructor should introduce students to the problem they'll work with in this project: search. Here's some material an instructor might use to introduce search:

- Introduce the general problem

We'll be working on a problem called "searching" for this project. Searching is one of the most studied and most important problems in computer science. There are some computer science researchers who believe that every problem in the entire field of artificial intelligence is, at its heart, a search problem. And, there are more search problems than even that!

- Define the problem

Ask students if they've ever searched for anything. Ask for examples of searching and write them up. If ideas slow down, ask students to give examples of searching that they haven't been involved in. A good example for the instructor to throw into the mix might be robotic search and rescue: a robot moves around a disaster site trying to locate injured or trapped people (about a half dozen prototype search and rescue robots were actually sent to Ground Zero after 9/11). You might also throw in some search problems that are very non-physical, like searching for a class schedule that fits with your schedule and meets the prerequisites you need or finding a great move in chess. When you feel you have enough suggestions, draw connections among the suggestions, focusing on the fact that search involves: exploring a space (a physical space or another kind of space) from a starting point or points and trying to find a goal.

- Introduce the specific problem

Explain to students that they'll be solving mazes and how mazes relate to the general problem. Solving a maze is one example of searching. We start at some point in the maze and explore the pathways of the maze until we find the exit. So, there's a space, the pathways of the maze; exploration of that space, by moving; a starting point, the entrance; and a goal, the exit. By learning how to search a maze, you'll also learn how to solve many other search problems. Try to connect to examples the students brainstormed.

StarLogo Environment Introduction

The instructor should bring up StarLogo on a screen that all students can see and demonstrate its parts: the programming window (and language), the graphical display of the "world," and the interpreter.

Closer: group viewing of maze solvers

Bring the students together and show a few groups' solve-my-maze procedures running on their mazes. This should be a chance to show off success!

Units 2–3 Materials: ifs, output/stop, and variables

Opener: Toolbox Procedures

Ask students how easy/hard it was to solve their mazes on the first day. Ask what commands they would rather have had (instead of turn-right, turn-left, step-forward, etc.) to control their maze explorers to make solving a maze easier. Record their suggestions. Note: Ensure that forward-to-junction, forward-to-wall, and at-junction? make it onto the list! Tell students that we will implement these commands.²

Work with students to define the **preconditions** — the set of conditions or requirements that must hold before executing their commands — and **postconditions** — the set of results and ensuing conditions caused by the commands — of each command. This might work well as a group exercise: go through one or two commands as a whole class, assign commands to the groups, have them write pre/postconditions, and come back together to present. You might use an example of a contract to motivate preconditions and postconditions: the subcontractor in charge of plumbing in a building might have as a precondition that the building structure is in place and has holes in the right spots and as a postcondition that the structure is unchanged (hasn't been demolished or built up an extra floor) but is plumbed and ready for toilets and water fountains.

Opener: ifs and ifelses

To motivate ifs and ifelses, write *at-four-way?* as a group. Then, write *at-two-way?* as a group. Start by telling students what *at-four-way?*'s postcondition is: it says "yes" if the turtle is at a four-way intersection, that is all four directions are open, and "no" if it is not.

Ask students if we can write this with the commands we know so far. (No!) Ask what's missing. (Something like "making decisions," "choices," "doing different things depending on the maze," or any other way of saying this.) Remind them that this was one difference between the ENIAC and the Pentium II (or the Analytical Engine) on the first day: the ability for the computer to decide what to do next based on what it has done so far. Tell the class that the commands that make decisions are called **control flow** commands.

Now, tell everyone that they're going to design control flow for StarLogo. Have the students get in their groups and *make up* a command that will let them write *at-four-way?*. Remind them that a programming language has to be made up of precise statements that the computer can understand; so, they should think of a name for their command and rules for how it can be used and what it does.

After a few minutes, bring the students back together and have a few groups write up their programs and describe their rules. Then, write up the StarLogo version and describe its rules:

```
to at-four-way?
  ifelse (blocked-towards? "front") or
        (blocked-towards? "right") or
        (blocked-towards? "back") or
        (blocked-towards? "left") [
    say "no"
  ] [
    say "yes"
  ]
end
```

Have students compare and contrast the StarLogo version to the different groups' versions.

Opener: variables

Now let's try *at-two-way?*! Draw a small maze and then have the class get into small groups briefly and come up with a definition for *at-two-way?*. Tell them to be prepared to apply their definition to the maze. Settle on a definition as a class when they come back together (using the maze). Hopefully, there will be at least two definitions: the exhaustive list of possibilities (front blocked and right blocked but not left or back; front and back blocked, but not left or right;

²StarLogo uses "command" to refer to functions that are predefined and "procedure" to refer to functions that are programmer-defined.

right and back blocked, but not left or front; etc.) and the counting solution (exactly two of the four directions are blocked). If there aren't try to draw these two definitions out.

Start with the exhaustive definition. Ask the class if we can do this in StarLogo yet. (Yes, but it's complex.. we need to check lots of conditions. How many? There are four directions and we need to *choose* two of them to block. That's $4C2$ or $\frac{4!}{2!2!} = 6$ different conditions to check, each one of which must check the **blocked?** value of all four directions!)

Now, ask if we can do the counting solution. (Again, technically yes, but counting with just ifs is even *more* difficult! If students don't believe this, have one or two people try.)

Now, like with **at-four-way?**, have everyone get in groups and decide out how *they* would tell the computer to do **at-two-way?** so that it matches the counting definition the class came up with, *but* when they need to make decisions, they must use StarLogo's if command. That is, they only get to make up commands for things that can't yet do in StarLogo. If on the last exercise any group just looked ahead and used StarLogo's commands, remind that group that when we compare their solution against StarLogo's it won't be very interesting unless they imagine their own solution.

Now, bring them back together and have some groups write up their programs. Meanwhile, write up StarLogo's version:

```
to at-two-way?
  let [ :number-of-paths 0 ]

  if not blocked-towards? "right" [
    set :number-of-paths (:number-of-paths + 1)
  ]
  if not blocked-towards? "back" [
    set :number-of-paths (:number-of-paths + 1)
  ]
  if not blocked-towards? "left" [
    set :number-of-paths (:number-of-paths + 1)
  ]
  if not blocked-towards? "front" [
    set :number-of-paths (:number-of-paths + 1)
  ]

  ifelse :number-of-paths = 2 [
    say "yes"
  ] [
    say "no"
  ]
end
```

Explain the rules to the StarLogo solution; tell them that “number-of-paths” is a **variable**. It has a value that we can change to keep track of something. After making the first four decisions, number-of-paths's value is the number of unblocked directions to travel. Again, compare and contrast solutions.

Opener: outputs (and stops??)

Now ask how hard it would be to write **at-four-way?** with the counting solution. (Almost exactly the same as **at-two-way?**!) Tell them to keep this in mind as they go through the activities.

Opener: summary

Summarize what has been discussed. (There's a lot!) “Control flow” lets their programs make decisions about what to do next. StarLogo's control flow uses the if and ifelse commands. Variables let programs keep track of values; StarLogo variables are created with the let command, have names starting with a colon (:), and can be set to values using the set command.

Activities

In the activities for this unit, students will need to have a way to find out which direction the turtle is facing. Refer them to the project web's "Turtle Language Reference" page, which includes an entry for command `turtle-direction`.

After students write `at-two-way?` and `at-four-way?`, they are guided towards noticing that the two procedures are very similar. (In fact, they can be written such that they only differ in one line, where the number of unblocked directions counted is compared against a constant.) They are next asked whether they can think of a more general procedure that could contain the code common to these two procedures. The intent is for them to arrive at a "helper" procedure that outputs (returns) the number of unblocked directions, e.g., `unblocked-directions`. Using such a procedure, rewriting `at-two-way?` becomes as simple as...

```
to at-two-way?
  ifelse unblocked-directions = 2 [
    output true
  ] [
    output false
  ]
end
```

(FYI, the above procedure can actually be written even more concisely, in one line: `output (unblocked-directions = 2)`. The comparison is executed first, and the result is `true` or `false`. This result is then used as the output value for the procedure.)

The point of writing procedure `unblocked-directions` is not just to rewrite the two previously written procedures. Writing `at-junction?` is much easier to write using `unblocked-directions`, and the activities ask the students to do so. The key observation is that a turtle should have at least *three* unblocked directions when at a junction: one being the way it came, and at least two more to have a choice of direction in which to proceed.

Closer: group viewing of maze solvers

As on day one, bring the students together and show a few groups' `solve-my-maze` procedures running on their mazes. This time, also show the difference between the code they wrote for day one and day two. Hopefully this will show off how much easier it is to do this with their new tools and how much more comprehensible the code is; might also change a maze just a bit and see if it's easier to change day one or day two code to handle the new maze. Point out that ease of coding, comprehensibility, and ease of maintenance are three of the reasons to use procedures.

Unit 3 Materials: Recursion

Write up the list of commands students brainstormed for their toolbox yesterday. This time, ask about the **forward-to-wall** command. Can we write this command yet? (No.) Why not? (We can't do something over and over again.) To write **forward-to-wall** and some other commands, we will need to instruct the computer to perform some action repeatedly!

Let's look at how we do that in StarLogo. If we want a command to execute over and over again, we simply include an instruction that starts the command over again as it is ending. Let's say we just wanted the turtle to move forward forever. (Write up code for **forward-forever** using recursion and show example.)

```
to forward-forever
  step-forward
  forward-forever
end
```

When a command executes *itself* like this, we call the command "recursive" and the process "recursion." We can use recursion to tell the computer to execute a series of instructions and then go back and execute them again. This opens up a whole new world of possibilities! Before, we just gave the computer commands and it executed them, one after another. Now, we can command the computer to repeat part of our program!

But, is this what we wanted? Does this make the turtle move forward to a wall?? (It *does* move forward to a wall, but then it just keeps trying to move.) Discuss the fact that stopping both movement and attempts at movement was implicitly a postcondition of the command (unless students actually listed it as a postcondition!). That is, we would like the command to stop running once it's done!

What we really need is a command that tells the turtle to move forward for a while but then stop once it hits a wall. We need to describe a condition under which the computer should stop making the turtle move forward. Does anyone know how we can decide when to stop? (The **if** and **blocked?** commands.)

So we do know how to do something only if a condition holds. So, instead of always starting that **forward-forever** command over, we can only start it over if there's no wall ahead. (Have the students get into groups and write a **forward-to-wall** command that stops when it reaches a wall. If a group finishes this quickly, have them start thinking about the day's exercises. After a few minutes, bring them back together and have them share solutions.)

Unit 4 Materials: Parameters

Opener: Real-Life Instructions with Parameters

Bring some spices and measuring spoons to today's activity.

Revisit day two's activity, building up a toolbox of commands, this time brainstorming commands for cooking. As students suggest commands, push them toward parameterized commands. Make sure that "Measure X teaspoons of Y" ends up as one of the commands. Have a student or two physically implement the command with the measuring spoons and spices, encouraging different possible implementations (e.g., "Measure 3 teaspoons of Curry Powder" might be three teaspoons or one tablespoon in implementation). Letting it get a bit messy might be fun (like measure it onto the table :).

Point out that these commands that perform differently depending on the extra information you give them are "parameterized" and that the individual pieces of extra information are the "parameters."

Opener: Parameters with StarLogo and Mazes

Bring up the list of commands students suggested on day two again, and this time emphasize which commands could be parameterized. Pick one and implement it to demonstrate StarLogo parameter syntax. Might also write the declaration for a "Measure X teaspoons of Y" command without actually writing the body. Discuss how parameters interact with preconditions/postconditions.

```
to measure-spice :n-teaspoons :spice-name
  ...
end
```

Closer: group viewing and generalization

As on day one, bring the students together and show a few groups' "solve-maze" functions running on their mazes. This time, also show the difference between the code they wrote for days one, two, and three. Hopefully this will show off how much easier it is to do this with their new tools and how much more comprehensible the code is; might also point out how much less code you have to write with parameters and ask what would happen if we changed the name of some maze explorer commands in each version (many more changes needed on day two, changes grouped on day three). Point out again how parameters help with ease of coding, comprehensibility, and ease of maintenance.

Next, run one team's maze solver on another team's maze. (Total failure, hopefully funny! :) Ask students to think about what happened as a lead-up to day four.

Unit 5 Materials: Algorithms

This day introduces the material that is central to what computer science and programming is: the algorithm. The algorithm is a concept used to understand how computers work, a tool used in making computers perform tasks, and a habit of mind for programmers and computer scientists (algorithmic thinking).

Prep for the day by taping out two mazes on the floor large enough to walk through, hiding one of them (by, e.g., covering it with a blanket), and getting the “red cube” hat (something to identify the wearer as the turtle from StarLogo).

Opener: What is an algorithm?

Tell students that today they’ll work with this concept “algorithm” which is central to computer science. (Use something like the text above, perhaps?) You might also use a dictionary definition. Here’s Merriam-Webster’s definition for algorithm: “a step-by-step procedure for solving a problem or accomplishing some end especially by a computer.”

Now, bring out the red cube hat and point out the taped maze. Have a student or students give instructions to guide you through the maze. Go through the maze in response to those instructions (be liberal in your interpretation, but make them be at least “conversationally” precise).

When done, ask students whether those instructions form an algorithm. Refer back to the description above (e.g., “Is it step-by-step?”, “Does it solve a problem?”, “What is the problem?”).

Now, reveal that there is another maze (but don’t reveal the maze!) and ask students if they can give one algorithm to solve both the maze they’ve already seen and the one that is hidden. Have students break back up into groups to think about it for a few minutes. Stress that they should try to figure out a very simple algorithm.

Finally, bring students back together, reveal the maze, and try out some algorithms. Discuss, again, whether they’re algorithms and what sort of problem they solve, etc. Point out that this whole discussion of “algorithm” has never mentioned a programming language. Algorithms are language independent.

Opener: How do I “implement” an algorithm?

Describe implementation of algorithms to students.

“Implementing” an algorithm means writing a program that executes that algorithm. Put the red cube hat back on and ask a group to implement their algorithm with you as the computer. This time, be adversarial in your understanding of their instructions. Once things have gotten sufficiently silly and surprising (walking through walls, turning continually until someone tells you to stop, etc.), stop the exercise and ask students why this isn’t working. The (an?) answer is that there is no precise, agreed-upon language.

When we implement an algorithm, we have to translate our step-by-step procedure into a program using a particular programming language. Tell students that they’ll now have a chance to design their own algorithms and then implement them in StarLogo’s programming language.

[Real-world examples: haven’t we already done this?]

Closer: remind of deliverables

Make sure the students remember that they need to have their first-draft written algorithm ready by the next Day.