Computer Science 50
Fall 2009
Scribe Notes

Week 0 Friday: September 4, 2009
Andrew Sellergren

Contents

1 Introduction (0:00–5:00) **2**

2 What Is Programming? (5:00–16:00) **2**

3 Algorithms (16:00–25:00) **3**

4 Scratch and Programming Constructs (25:00–72:00) **4**

Computer Science 50
Fall 2009
Scribe Notes

Week 0 Friday: September 4, 2009
Andrew Sellergren

1 Introduction (0:00–5:00)

- This is CS 50.
- Lectures are available in MP3, Flash, and QuickTime formats here.
- Shuttleboy Voice! Soon you'll be able to call 617-BUG-CS50 and find out the next shuttle time!

2 What Is Programming? (5:00–16:00)

- One student offers: "Programming is telling the computer what to do." This is true, although we don't mean at the high level as in point-and-click, but rather providing a set of instructions so that underneath the hood, the computer can operate.
- Our first foray into programming will be in C and is meant to underwhelm you:

```
#include <stdio.h>

int
main(int argc, char *argv[])
{
    printf("hello, world!\n");
}
```

  Don't worry right now about what all this syntax means–in fact, it's not very intellectually interesting. Suffice it to say that we have to follow a set of conventions in order to get the program to work. And, indeed, if we run `gcc`, a C compiler, on our `hello.c` file, we get no error messages (whew!) and we get a working program named `a.out`. Now if we run the command `./a.out`, we get the message `hello, world!` printed on the screen. Our first program!

- As intended, this program is fairly underwhelming. Especially when compared to something like this. Perhaps now you can understand why we want to start you off with Scratch as opposed to C!
- This Scratch project was made by a former TF, but don't think that only TFs have the skill to make something entertaining like this. Check out this project.
- So how do we go about making a program like this? The basic building blocks of programs like this are algorithms. An algorithm is simply a set of instructions which the computer can understand.

3 Algorithms (16:00–25:00)

- Let's say we want to come up with an algorithm for putting our socks on in the morning. We'll write this algorithm in pseudocode, a not-quite programming language that allows us to express human-readable instructions that can easily be translated into real code. Our socks algorithm looks like so:

```
let socks_on_feet = 0
while socks_on_feet != 2
    open sock drawer
    look for sock
    if you find a sock then
        put on sock
        socks_on_feet++
        look for matching sock
        if you find a matching sock then
            put on matching sock
            socks_on_feet++
            close sock drawer
        else
            remove first sock from foot
            socks_on_feet--
    else
        do laundry and replenish sock drawer
```

- In step 1, we set a variable `socks_on_feet` to the number 0.
- In step 2, we enter a loop that will be repeated as long as the condition (`socks_on_feet` does not equal 2) is true.
- Inside the loop, we open the sock drawer, look for a sock, and then enter a condition:
  - If a sock is found, put it on.
  - Otherwise, do laundry and replenish sock drawer.
- Note that our code is indented in key places. Everything which is encapsulated in the `if` condition is indented to indicate so. The computer most likely doesn't care about this indentation but it makes the code easier for us to read.

- This code has a bug. If only one sock is in the sock drawer (or none of the socks has a matching partner), it will remain in the while loop forever.
- This is called an infinite loop. As you will see later, sometimes programs we write will contain infinite loops. We will recognize these cases because when we run the code, the computer will "hang"–it will keep running and produce no output because it is stuck in a loop.
- It might seem obvious to us what to do in the case where there is only one sock in the drawer, but a computer cannot make assumptions the way we can.
- Therefore, our algorithms must be extremely specific and precise and account for all possible cases. Nothing can be assumed.
- Not only must our instructions be precise, but also the language in which they are written must be precise. That is, we must follow the guidelines of syntax for whatever programming language we write in. As we saw earlier, the GCC compiler successfully created an executable program based on the lines of code we wrote. If we hadn't followed the syntax guidelines of C, then the compiler would have spit out error messages.
- What exactly does the compiler do? It translates our human-readable instructions into machine-readable binary. Compilers are often operatiing-system-specific, which is why you must buy a different version of Microsoft Office for Macs and PCs. One of the appeals of the Java programming language is that it's *cross-platform*. The Virtual Terminal Room (VTR) we use for Virtual Office Hours (VOHs) is written in Java because the company who created it (Elluminate) decided it would be more cost-effective than writing two separate versions. There are, as you might expect, downsides to Java, however.

4 Scratch and Programming Constructs (25:00–72:00)

- Problem sets will be released on Fridays at 7 p.m. We encourage you to start early so that you can take advantage of the many office hours during the week.
- To start working with Scratch, you'll need to download the program from MIT's website.
- Once you install Scratch and open it, take note of the following layout:
  - On left, notice the puzzle pieces, which represent statements. Programs will be composed by putting puzzle pieces together in a particular order.
  - At bottom right are sprites, or characters that will carry out your instructions.

  - At top right is the stage, where the program will be carried out.
  - To the left of the stage is the scripts area, where puzzle pieces must be dragged and strung together.

- We can recreate our very simple C program in Scratch using the "say" block. `Hai1.sb` is equivalent to `hello.c`, only a little more colorful.

- If we use the "ask" block, we can take the user's input, which will be stored in the `answer` variable, and repeat it back.

- Obviously, we're taking baby steps, but realize that that's what programming is all about–taking very basic building blocks and creating functions and more complicated programs.

- `Hai2.sb` is slightly more complicated. The cat will say "O hai, world!" for 1 second, wait 1 second, say it again for 1 second, wait 1 second, and say it again for 1 second.

- So far we've only made use of *statements*, which are direct imperatives given to the computer. But if we want to introduce logic into our program, we'll need *boolean expressions* and conditions. Boolean expressions are those that have only two possible values: true or false, yes or no, on or off, 1 or 0. No matter how you say it, it's a simple variable. Based on these expressions, we can take different courses of actions using if-then statements. You can also nest these conditions so you can take more than two courses of actions.

- `Hai4.sb` and `Hai5.sb` make use of conditions and boolean expressions. In the first, the condition `1<2` always evaluates to true, so the cat meows everytime we click the green flag. In the second, however, the condition says, "pick a random number between 1 and 10 and if that number is less than 6, have the cat meow." This is what we call a pseudorandom number generator. Although it seems simple here, the idea of forcing a deterministic machine to non-deterministically generate a random number is actually quite complicated. There is an entire branch of study devoted to this very task. One theory holds that random numbers can be generated from the white noise picked up by a microphone. In any case, the effect of this pseudorandom number generator on our program is that the cat will meow approximately half the time we click the green flag.

- If we want our cat to meow multiple times, we can certainly just duplicate the statements however many times we want. But this has several disadvantages:

  - It is resource-inefficient.
  - It is tedious.
  - It makes it difficult to change what the cat is saying.

Remember our goal is not just to accomplish a task, but to accomplish it elegantly and efficiently.

- To that end, we can use loops when we wish to repeat a statement. In `Hai6.sb`, we implement a loop which causes the cat to meow indefinitely. In `Hai7.sb`, we combine a loop and a condition so that the cat will meow only if we the mouse pointer is touching it or, in other words, if we are petting it. In `Hai8.sb`, we add an extra condition so that the cat will meow indefinitely, but will roar if we touch it with the mouse pointer.

- *Variables* are another useful programming construct. They allow us to store information about the *state* of a program. For example, in the case of the socks algorithm, the variable `socks_on_feet` stored the number of socks Josh had on his feet. In `Count1.sb`, we set the variables `counter` and increment it as the program runs. What problem could we run into? If `counter` is only stored with 8 bits, then the maximum number that can be stored in it is 255. No matter how many bits it is stored in, there will be a maximum number that it can represent.

- `Count2.sb` is similar in spirit to `Hai5.sb`. Using a pseudorandom number generator, the sheep will make its noise only about half the time.

- *Arrays* are essentially collections of related variables. In the game `FruitcraftRPG.sb`, for example, an array is used to store the different types of fruit which have been collected.

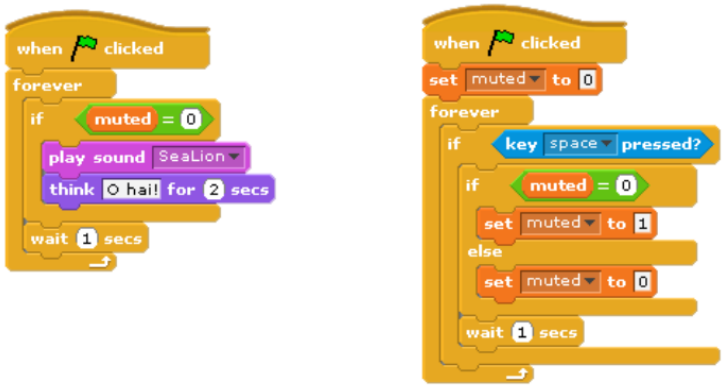
- Now is a good time to introduce the concept of *threading*. This is a fairly complicated concept which doesn't usually get introduced in the first week of a computer science course, let alone in the first semester of computer science training. However, let's Scratch the surface of a threading discussion.[1]

- Threading refers to the notion of multiple threads of code executing simultaneously. Formerly, when CPUs contained only a single core, multithreading was actually just an illusion: the CPU would switch back and forth very quickly between processes so that the computer would appear to be doing multiple things simultaneously. However, with the advent of multicore processors, computers really can execute multiple instructions simultaneously so long as the program has been coded this way. In Scratch, threading is again an illusion. Two sprites moving simultaneously is not *really* two sprites moving simultaneously, but rather Scratch moving one sprite and switching very quickly to move the other sprite, and so on.

- In `Move1.sb`, we have a simple animation of a duck moving back and forth across the screen, screaming and turning around every time it touches the edge. This is a single thread.

[1] Yes, yes I did just make that joke.

- In `Move2.sb`, we achieve "multithreading," at least in appearance. In terms of programming, we have two different scripts associated with two different sprites, a cat and a bird. For the cat, we begin by placing him in a given spot on the stage and orienting him in a random direction. Then we begin a loop whereby if he touches the bird, then the game ends; otherwise, orient toward the bird and advance one step.
- For the bird, we again place and orient him and then move him around the stage three steps at a time. Effectively, then, the cat is chasing the bird until he catches him.
- In `Hai10.sb`, we show another example of two scripts interacting with each other, this time by means of an explicit variable named `muted`:



  As you can tell from the code, the righthand script checks constantly or "listens" to see if the spacebar has been pressed. If it has been pressed, then the `muted` variable will be set to true and the sound introduced by the lefthand script will stop playing.

- Try going through `David.sb` and dissecting the code which creates a boxing match between you and David!
- *Events* are another method of communicating between sprites. `Marco.sb` leverages events to play the game of Marco Polo.
- Scratch also offers sensor boards which take user input in the form of sound, light, and movement, as demonstrated by `singer.sb`, `Masquerade.sb`, and `davidwu.sb`, respectively.
- Check out `Oscartime.sb` for another example of what you can do with Scratch. And check out `Dragon-drop.sb` for an excellent demonstration of dragon-drop programming!