Computer Science E-1
Spring 2010
Scribe Notes

Lecture 9: April 19, 2010
Andrew Sellergren

Contents

Computer Science E-1
Spring 2010
Scribe Notes

Lecture 9: April 19, 2010
Andrew Sellergren

1 Announcements (0:00–1:00)

- Next week, April 26th, during normal lecture time, will be Exam 2. After lecture tonight, TF Alex Chang will hold a review session in lieu of section. Don't worry, it will be filmed!
- On Wednesday, we'll post the next assignment as well as the specification for the final project which will culminate the course with a web design venture.

2 Programming (1:00–105:00)

2.1 Introduction (1:00–9:00)

- Programming is the act of writing a set of instructions which a computer can understand and follow. Programs might be e-mail clients, web browsers, document editors, or any number of other types of software. More generally, programming is about thinking logically and methodically. As sophisticated as you might imagine computers are, they can only understand the most basic of instructions. If you fail to specify exactly what a computer should do in a given situation, it might crash or hang or otherwise behave unpredictably.
- Computer programs can be written in many different languages, among them Java, C, C++, C#, BASIC, ADA, Python, Perl, PHP, R, COBOL, and FORTRAN.[1] In order for a computer to understand any of these programming languages, they must be *compiled* or converted into binary. Recall that the CPU accepts a limited number of instructions which are encoded by specific patterns of zeroes and ones. The *source code* that you've written in one of these programming languages will be translated by the *compiler* into nothing more than these CPU instructions which are as simple as addition and subtraction.
- Different *architectures*, that is, different brands and versions of processors, require different compilers because the specific patterns of zeroes and ones that encode instructions vary between them.
- As with a spoken language, it would be impossible for us to teach you an entire programming language in a single night. However, we will get a hold on a graphical language called Scratch which was developed at MIT to allow novice programmers to visualize the low-level logical building blocks that combine to create all programs.

[1] Languages like HTML, CSS, and XML are not quite programming languages, per se, since they deal with aesthetics and markup more than logic.

2.2 Hello World Programs (9:00–25:00)

- Before we dive into Scratch, let's take a look at a very basic C program, whose only function is to print the words "hello":

```
#include <stdio.h>

int
main()
{
  printf("hello");
}
```

  The line that begins with `#include` tells the compiler to go fetch a file called `stdio.h` which contains some code that someone else wrote that we want to use in our program, including a *function* called `printf` which allows us to print to the screen. `printf` takes a single *argument*, or input,[2] which is the text to be printed. Note that the curly braces, the parentheses, and the semicolons are all just annoying syntax that unfortunately are required. These syntactic conventions are one of the reasons why we prefer a language like Scratch which hides these details from us.

- Once we write these lines of code in a text editor and save it to a file called `hello.c`, we can compile it by running the command `gcc hello.c` from the command line. By default, this will output an executable file named `a.out`. If we then execute the command `./a.out`, we see that "hello" is printed to the screen!

- GCC is the compiler that we mentioned earlier. It is the middleman that translates our C source code into zeroes and ones that the computer can actually understand and execute.

- Some of the programming languages we mentioned earlier—PHP, for example—are so-called *interpreted* languages, which means they aren't compiled ahead of time but rather on the fly when they are executed.

- Let's take a look at a simple program in Java:

```
class Hello
{
  public static void main (String [] args)
  {
    System.out.printf("hello");
  }
}
```

[2] Actually, it can take multiple arguments, but only the first is required.

Although the syntax is quite different, this program does the exact same thing as our C program above. To compile it, we run the command `javac hello.java` and to execute it, we run the command `java Hello`.

- Generally, this type of program is called a "Hello World" program and is the first code you'll write in a new language. Take a look here for examples of "Hello World" programs in many different languages.

- To execute programs in an interpreted language, we make use of an *interpreter* rather than a compiler. Once we've written our source code, we can immediately execute it by passing it to the interpreter. There is no separate step of compiling. Let's take a look at "Hello World" in PHP:

```
<?
  printf("hello");
?>
```

To run this program, we execute the command `php hello.php`, which passes our source code to the PHP interpreter.

- Question: is it possible to edit the binary after a program has been compiled? Yes, in fact, this is what hacking a program often involves since the original source code isn't available.

- The difference between compiled and interpreted languages is much more evident as programs become more complex. Compiling a program that is millions of lines long takes more than a few seconds. However, compiling a program ahead of time means that the program will generally execute faster than the same program written in an interpreted language which must be interpreted each time it is executed.

2.3 Loops (25:00–32:00)

- If we wanted to print "hello" to the screen 10 times, we might choose to copy and paste the same line of code 10 times in a row. Even knowing nothing about programming, you can recognize this approach as inefficient. Instead, we can use what's called a loop. Take a look at this program in C:

```
#include <stdio.h>

int
main()
{
  while(2 < 3)
  {
    printf("hello");
```

```
  }
}
```

The expression `2 < 3` is a *condition* which, as long as it is true, will cause the code between the curly braces of this `while` loop to be executed. Since 2 is always less than 3, this code will execute indefinitely. This is called an *infinite loop* and, as you might've guessed, is bad programming practice. You may have experienced it, however, in a program in which execution seems to grind to a halt and all you see is a spinning beach ball or a timer.

- Of course, loops are much more useful when the condition contains a *variable* whose value might change as the program continues to execute. In this way, the code in the loop will not execute infinitely, but only until the condition evaluates to false.

- Let's take a look at another example of an infinite loop, but one that's slightly more useful:

```
#include <stdio.h>

int
main()
{
  int x = 1;
  while(x > 0)
  {
    printf("%d\n", x);
    x = x + 1;
  }
}
```

Here, we declare `x` as a variable and initialize it to 1. Next, we execute the code in the loop since at this point x is greater than 0. Finally, we update the value of `x` by adding 1 to it. At this point, we return to the beginning of the loop and check the condition again. Since `x` is still greater than 0, we execute the loop again. Actually, `x` is always going to be greater than 0 since we're going to increment it. The actual code of the loop prints out the value of `x` using some cryptic syntax. Ultimately, then, this program counts upwards from 1 indefinitely.

2.4 Algorithms (32:00–48:00)

- To accomplish any task in programming, we must come up with an *algorithm*, or a set of steps. Generally, a problem can be solved with many different algorithms, but ideally one will be more efficient and less error-prone than the rest.

- Imagine that we want to look up a name in the phonebook. One algorithm for doing so would be to open to the first page, look for the name, and then flip to the next page and the next and so on. To be more efficient, however, we might consider successively cutting the problem in half. Let's say the name begins with a B. We would begin by opening to the middle of the phonebook, to the section beginning with M and recognizing that B comes before M in the alphabet. So we throw away the second half of the phonebook, and focus on the first half. We then take the first half of the phonebook and cut *it* in half, arriving at the F section. B comes before F, so we again throw away the second half. We continue to do this until we arrive at the B section. If our phonebook is a million pages long, it'll take a maximum of 20 steps to find any name.

- 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. Notice that our variable is descriptively named so that we will always know what it

refers to when it appears anywhere in our program.

- 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.
- 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 and increment our variable.
  - Otherwise, do laundry and replenish sock drawer.
- Inside the first if statement, there is a second if statement. Once we've found a single sock, we must check for a matching sock. If a matching sock is found, then we put it on.
- 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 another example of an infinite 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.
- A few other problems or at least points that need to be clarified: the code assumes that *a*) when you take a sock off, you put it back in the drawer, *b*) you have exactly two feet, and *c*) when you have one sock on, you will put another sock on your other foot, not the same foot.

2.5 Scratch (48:00–105:00)

- Scratch was designed by MIT to target middle-school-aged children in after-school community centers, but we've been using it for a few years now in CS 50, Harvard College's introductory computer science course, as a way of introducing students to programming without overwhelming them.
- 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. The default sprite is a cat.
  - 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.

- Although this programming environment may seem elementary, it can be used to create some pretty sophisticated and fun programs like Scratch Scratch Revolution!, created by a former CS 50 student.

- In Scratch, the puzzle pieces are color-coded according to what type of programmatic statements they encompass. The Control statements, for example, are orange in color and include the "when [green flag] clicked," piece, which is similar to the `main` statement in our C programs from earlier. Take a look at this "Hello World" program in Scratch:

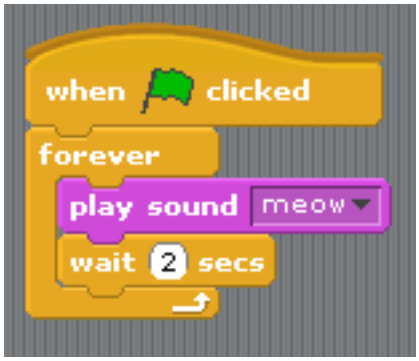
- In addition to printing text to the screen, we can paint, play sounds, pause execution, and do many other things very easily using Scratch. Take a look at this slightly more complex program:

  These statements which we've conveniently snapped together will execute top to bottom, one after another.

- Let's try empowering the cat to sound like a cat:

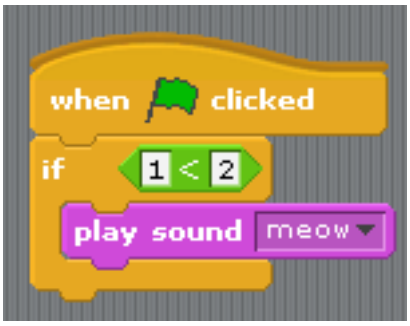
- If we want our cat to meow multiple times, we can certainly just duplicate the statements however many times we want. But this has several disadvantages:
  - It is resource-inefficient.
  - It is tedious.
  - It makes it difficult to change what the cat is saying.

  Remember our goal is not just to accomplish a task, but to accomplish it elegantly and efficiently. Let's try using one of the loop constructs provided by Scratch:



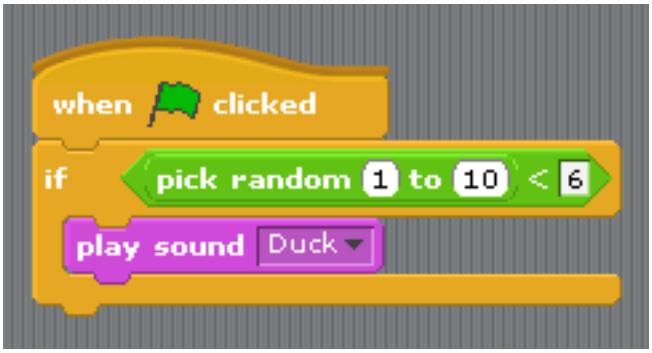
  The "forever" block at first glance appears to only allow one puzzle piece inside it, but you'll notice that it will expand to fit however many pieces you want as soon as you drop them there.

- Let's begin using if statements as we did with C earlier:



In this case, because 1 is always less than 2, the cat will always meow when we click the green flag. Notice that the if condition has a diamond shape. This is to help us recognize what kind of expressions can be placed inside it. If we click on the green Operators page, we see that the arithmetic operations like multiplication and division are oval in shape. This makes sense because the statement "if 1 times 2," has no real meaning. The comparison operators, on the other hand, including less than, greater than, and equal to, all evaluate to true or false. These are so-called *boolean expressions* and, as their diamond shape indicates, they can be plugged in as if conditions.

- If we want to add a bit of dynamism to our program, we can use Scratch's built-in random number generator like so:

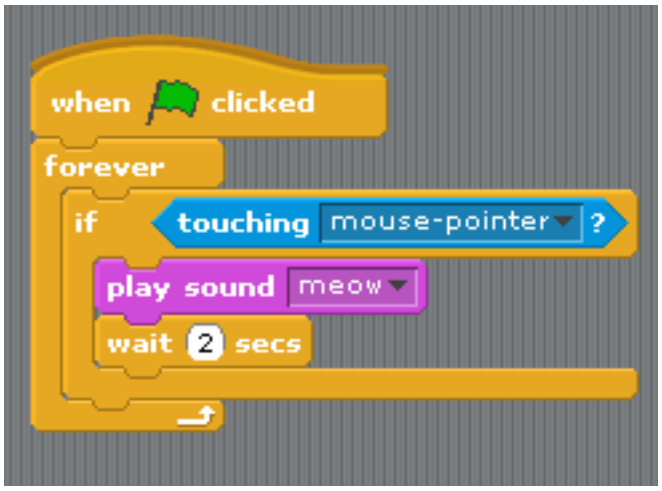


Here, we're effectively simulating a coin flip. Since the random numbers that Scratch picks will be evenly distributed one through ten, about half the time the number chosen will be less than six. Thus, about half the time, the Duck sound will play. Incidentally, we enabled the Duck sound by clicking on the Sounds tab, clicking Import, and choosing it from the Animals folder.

- If you scroll down on the Operators page, you'll see more complicated if statements. As in other programming languages, we can write if–else

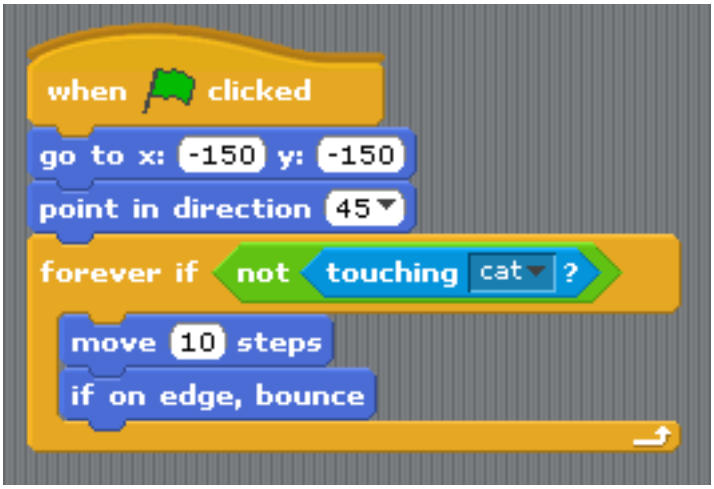
statements as well as if–else-if–else statements.

- In this next example, we add dynamism by sensing whether or not the user's mouse pointer is touching the sprite:

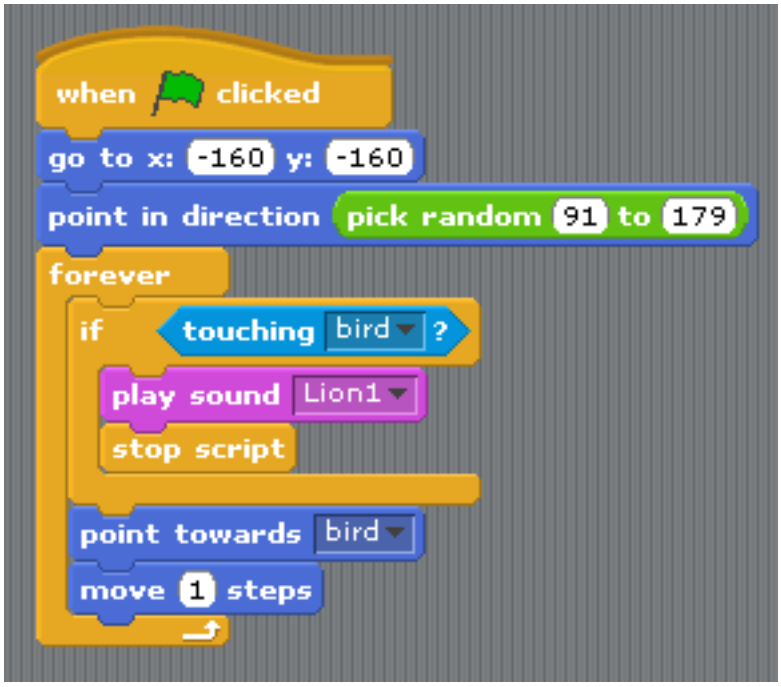


If we move the cursor over the sprite, it will meow.

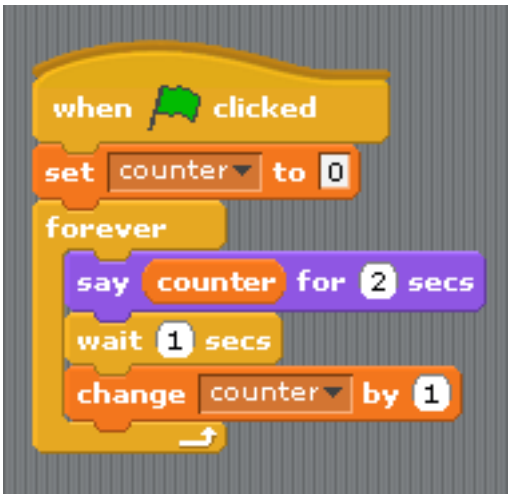
- With pieces from the Motion page, we can make our sprites move. David's slightly more complex example actually involves two sprites—a cat and a bird—which interact with each other. In fact, the bird moves around semi-randomly, bouncing off walls and the cat perpetually points in the direction of the bird and slowly advances on it. If the cat ever touches the bird or "catches" it, he will roar. Take a look at the bird's script:



And the cat's script:



- A key takeaway from this whole process is that you should take baby steps when trying to create any program. Don't sit down and wonder how you can implement Dance Dance Revolution. Rather, sit down and think, how can I make a single arrow image move to the top of the screen. Then continue building from there.
- We can create our own counting program in Scratch like so:



As you can see, we begin by initializing a variable named `counter` to the value 0. Then we announce the value and increment it by 1. We do this indefinitely for the life of the program. To create a variable, simply navigate to the Variables page, click Make a variable, and provide a name for the variable. Once you've provided a name, you'll be asked whether this variable pertains to all sprites or just the currently selected sprite. If it applies to all sprites, it is a so-called *global* variable. There are some cases in which you might want to limit a variable's scope to a single sprite. On the Variables page, the checkboxes next to the variables you've created indicate whether their values will be displayed on the stage while the program is executing. This can be useful for tracking the progress of your program.

- Another type of variable that we have yet to mention is called a "list" in Scratch although in other programming languages it is known as an array or a vector. This type of variable allows more than one piece of data to be stored with it. In Fruit-craft, a role-playing game programmed by a former CS 50 student, for example, the player's inventory of fruit is implemented as a list.

- Scratch programs are multi-threaded by nature. Just as Microsoft Word can execute many different programming instructions at the same time,—for example, to underline misspelled words even as you type—so too can Scratch handle two independent scripts at once.

- One other feature of Scratch worth mentioning is *events*. Sprites can interact with each other not only by touching, but by broadcasting and receiving pieces of data. Using events, we can recreate the game of Marco Polo whereby when we press the spacebar, one sprite will say "Marco!"

and broadcast an event. The other sprite listens for that event and says "Polo!" when she receives it.

- To see what you can do with Scratch, take a look at Oscartime, which David created when he had way too much free time!