Northeastern University
Dept. of Computer Science

DAY 1, LECTURE 1
"INTRO TO SCHEME, INTEGER VALUES AND FUNCTIONS"

Lesson Overview:

Students learn basic Scheme syntax (operator *operands*), and become familiar with numerical values and operators.

Learning Objectives

Students will:

- Understand format of Scheme statements
- Be able to declare numerical values and operate on them with simple functions
- Understand the distinction between functions and values
- Understand common Scheme error messages

Product Outcomes

- Students will write functions to solve simple mathematical equations
- Students will write code to calculate the next X and Y coordinate of the UFO

Ritual

- Students work in pairs, one coding while the other leads. Every ten minutes, we yell "switch!" and they must do so.

Length: 80min w/ 100min lab exercises (flexible)

Materials and Equipment

- ❒ PowerPoint Slide Deck
- ❒ Computers w/DrScheme Preinstalled
- ❒ Student notebooks or papers

Preparation

- ❒ On the board: What makes up a *language*?
- ❒ Write agenda on board

Vocabulary/Concepts (on board, but to be explained)

- Programming Language
- Value and Function

Lesson Plan:

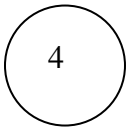
I. Introduction — Time: 10 mins

- ❑ What is a language?
  - A system of rules and symbols that expresses meaning
  - How many of you speak more than one language?
  - Do you find that you speak differently to your friends than to your teachers? How come? *Different languages are best suited to different tasks*. Some ways of speaking are better if you're going out with friends, and others are better if you're asking your mom to let you go out.
- ❑ Computers have their own languages too!

  - People are really good at context. If I ask you to go to the store and buy me something to drink, you can fill in all kinds of information that I didn't tell you. You can find your own way to the store. You can get yourself there, and find the drinks section. You might even know what I do and do not like to drink, and make a good choice accordingly.
  - Computers are terrible at context. They need every single thing spelled out carefully. When you talk to a computer, you need to be *very specific*.
  - However, once you tell a computer how to do something…it does it **very, very fast**. It also does it **exactly** the way you asked it to. If I asked you to go to the store and check all the drinks to find the one with the least sugar, it would take you a long time to compare the labels on every single beverage.
- To illustrate this, we're going to play a quick exercise. My friend here is a human, and he's good at context. I'm going to put this pen here on the desk. How would you ask him to pick up the pen? Now I'm going to be a computer…which means I need things to be really specific. How would you tell *me* to pick up the pen? (overact it to drive home the point.) What did we learn?
- *This* is where computers come in handy. Computers are powerful tools that we can master, as long as we learn to translate our context into specific instructions.
- Over the next days, you are going to learn how to speak to a computer using a language called Scheme. Your job will be to translate the context that comes so easily to you into a language that a computer can understand.

II. Introduction to DrScheme — Time: 5-10 mins

- Have students open DrScheme on their computers. Refer to PowerPoint or overhead projector as necessary.
- DrScheme is a tool that allows you to write Scheme programs. On your screens you'll notice two large boxes: the *Definitions* window at the top and the *Interactions* window at the bottom. For now, we're going to just focus on the Interactions window.
- The Interactions window is where you can "try out" different bits of Scheme code. You type in a Scheme program and hit Enter, and DrScheme will run the program and show you the result. If anything went wrong, it will spit out an error message.
- Write Circle of Evaluation on board, put in the number 4. "What is the value of everything in this circle?"
- Whenever you run a Scheme program, it also does an evaluation.
- Type in the number 4, hit return and see if DrScheme agrees with you. Congratulations: you just wrote your very first Scheme program, and it came out to be the same value as what you'd expected. Try typing in other numbers and see what happens. What happens if you write a decimal?

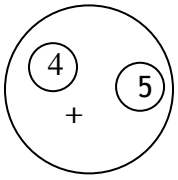


III. Nouns and Verbs, Values and Functions — Time: 5-10 mins

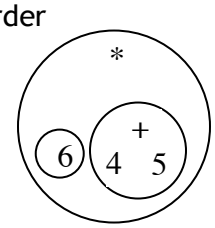
- Can someone explain what a noun is? A verb?
- What is the difference between the two?
  - I can pick up a "pen", but I can't pick up a "throw". However, I can throw a pen.
- Verbs are made to *do something* to nouns. *Break* a *toy*. *Drive* a *car*. *Clean* a *room*.
- When you apply a verb to a noun, you get some result. If you *clean* a *room*, you get a nice clean room. If you *break* a *toy*, you will have a broken toy.
- Scheme has similar concepts, called *functions* and *values*.

  - Values are things like numbers and words. When I asked you what was the value of the number 4, you knew that it's value was itself. In fact, all of the numbers you have been entering are all examples of values.
  - A Function is like a verb, and it works very much the same way. If you *apply* a function to a value, you will *produce* some result. Can anyone tell me what I'll get if I add 4 and 5?
- You already wrote programs that were just values. Now you're going to write programs that apply functions to those values.

IV. Reverse Polish Notation — Time: 10-25 mins



- Write 4 + 5 (mixed up) in Circle of Evaluation. What is the value of the stuff in the circle? Let's figure out the math first. (write the equation. We know we're adding, so we start with the plus sign. ("+" in the middle) Then we add 5 and 4 on either side. ("5 + 4"). Does it matter if I write (4 + 5)?
- Now use subtraction. Have kids argue, and then figure out that the order of inputs matters. Have them type examples into Scheme.



- Write 4 + 5 * 6 (mixed up). What is the value of the stuff in the circle? Have kids argue that order of operations matters. Write out two algebraic formulas. ("5*6 + 4", "4*5 + 6") Make a rule that no circle has more than one function in it.
- We need to improve our Circle of Evaluation! Let's use the two solutions we came up with: all circles have only function, and it matters what order the arguments are written.
- To read our Circle, we look at the function in the outermost circle. (Write the parens and the "*") In order to apply this function, we need to evaluate the left side and then the right. The left side happens to be *another* Circle of Evaluation (more parens), so we apply the same rules there: write the function and then look at the left and the right. These are numbers, so we can write them in directly. `( (4 + 5) * )` *Now* we can look at the right side of the outermost circle. This is just a Number, so can write that in. `( (4 + 5) * 6 )`
- Scheme works almost the same way, except that the functions come ***before*** the values. To use our example above: `(* (+ 4 5) 6 )`
- Try writing (+ 4 5) in the Interactions window, and see what you get. What did you end up with? Now try writing in the complex example we used.
- What are some other functions that you could try? (*, - and /)
- Who knows what an exponent is? What do you think `(expt 3 2)` will produce?
- Who knows what a **contract** is?
- Show the contract for + on the board, and have students volunteer the contracts for the other math functions covered so far. *Leave this written on the board.*
- Walk through at least three more examples, and have students come up with their own.
- Ask them to use paper for the Circle of Evaluation if they're having any problems. It is key to have them practice the actual order of evaluation that Scheme uses.
- **Mini Lab 1: Write a few simple algebraic statements on the board, and have students (1) draw the circles of evaluation for them and (2) write them in Scheme.**

V. Writing your own functions Time: 20 mins

- It is useful to think of functions as machines, which take something as input and return something else as output.
- You've all seen examples of functions in math class, right? y=2x, for example.
- This function is a machine that takes x as input, doubles it, and returns it as output y.
- Instructor: Draw a table, and have kids fill in additional x and y values.
- Imagine all possible x's in a big line, stretching out to infinity on one side of the board. Now imagine all possible y's, stretching out to infinity on the other side. The function y=2x is a machine that shows how to convert points from one side to the other. Write:

```
; y=2x : Number -> Number
; returns a Number that is double the input
```

- What I've just written is called a **contract and purpose statement**. Combined, they tell us almost everything we need to know about a function. Every function contract has three parts:
  - Name of the function (y=2x)
  - Input to the function (Number)
  - Output of the function (Number)
- The purpose statement is a simple explanation of what the functions *does*. If you had to describe the function to your mom, you'd say exactly what's in the purpose statement.
- Machines can work on more than just numbers though. Let's look at some other machines, and write contract for them as we go.

  - A CD player, for example, takes in CDs and outputs music.

```
; CD Player : CD -> Music
; Outputs the music from the CD
```

  - What is the input of a Coffee Machine, that mixes grounds and water to make coffee? The output?

```
; Coffeemaker : Grounds Water -> Coffee
; Mixes water and grounds to produce coffee
```

- Suppose you have to make the blueprints for a CD Player. You know that the input to this machine will be a CD, and you have to produce music. When you buy a CD player, does it only work with one CD? Wouldn't it suck if you had to have a different CD player foe every CD?
- *Machine blueprints talk about inputs, without having to specify what those inputs **are**.* A CD player just talks about *some* CD, but not any specific one. It refers to some "placeholder", which might be any CD.
- In our example of y=2x, the function doesn't talk about any specific x. It just talks about a placeholder number, which it will double and produce as y.
- In Scheme, functions talk about inputs using placeholders too. (open the Definitions window and start typing…)
- Suppose we want to make a function called double. Double takes in a number and doubles it. In my definitions window, I'm going to write that down so I know what I'm doing. In Scheme, you put a semicolon before any line that isn't code. So in my definitions window, I'm going to write "`; double takes a number x and doubles it.`"
- Draw table on the board.
- What is our input? (a Number) Our output? (another Number)
- (Use table on the board, "how do I double 5? 6? 7?" Circle repetitive code.)

- I've written lots of code, to do the same thing over and over. This is like using a new CD player for every CD: we have to write (* 2 …) for every number. This is crazy!
- We can look to our math function y=2x for inspiration. X is a placeholder, and the function works for any input!
- Go over examples, using test case boxes on our mythical "double" function.
  - Use Circle of Evaluation with `(double 4)`, `(double (+ 4 6))`
- Click run. What happened? (double isn't defined).
- Now we have to define the function. *Every function has two parts: the header and the body*.
- The header always begins with the `define` keyword, followed by the name of our function and the placeholder names of all the inputs. What is the name of our function? (double) How do you know (the contract!). How many inputs does it take? (one) How do you know? (the contract!)
- What did we name the input (x)? How do we know? (the purpose statement!)
- Now we can write out function body…which is *exactly* what we just wrote.
- Does it matter what we call the input? Instead of x, could I write q? Or z? (as long as the function header is changed accordingly).
- Click run to see if the test cases worked
- **Mini Lab 2: Have students write a function for the circumference of a circle.**
- **Mini Lab 3: Write a program to convert a temperature `C` from Celsius into Fahrenheit. `C->F : Number->Number`.**
  The formula to do this conversion is (9/5) * C + 32.

- **Mini Lab 4: Write a program to convert a temperature `F` from Fahrenheit into Celsius. `F->C : Number->Number`.**
  The formula to do this conversion is (5/9) * F - 32.

Lab 1.1

1. Evaluate the following Scheme statements in DrScheme, one at a time. Read and understand the error messages, and try to correct the statements so that no error messages appear. *Key point to remember: a function name must always follow an open paren.*

`(+ 5 (/ 1 0))`	`(define h(x)` `(+ x 10)`
`(define (g x)` `+ x 10)`	`(define (f 1)` `(+ x 10))`
`(sin 10 20)`	

From the File menu, select "Save Definitions". Save your work as "Day 1" on your disk.

VI. Game Code

From the File menu, select "New". Put the answers to the following questions in the Definitions window.

1. Suppose you have a UFO, which has an x and y coordinate. Write the coordinate function `update-x : Number->Number`. It's input is a number, representing the x-coordinate of the UFO. The output is the *next* x-coordinate of the UFO, which is always 3 more than the old x-coordinate.

2. Write the coordinate function `update-y : Number->Number`. It's input is a number, representing the y-coordinate of the UFO. The output is the *next* y-coordinate of the UFO, which is always 3 more than the old y-coordinate. Don't be lazy! Don't copy and paste from `update-x`! If you made a mistake in `update-x`, you don't want to copy the mistake!

Save these Definitions as "Game" on your floppy disk.