Reading Assignment: Chapters 1-3 of our on-line textbook.

___ Syllabus.

___ Roll call, find out who is/isn’t enrolled.

___ Overview of reading assignment: Our textbook takes a bottoms-up approach to Scheme’s computational model. Each of the first few chapters introduces a small portion of the computational model, explaining the syntax and semantics of each construct that is presented.

Ch. 1 Introduction

___ Syntax rules: These specify legal words, expressions, statements, or sentences of a language.
___ Semantic rules: These specify what each of the legal words, expressions, statements or sentences mean.

It is important to have an accurate mental model of the computations you ask the computer to perform.

We will learn to program a computer in a language called Scheme/Racket/Swindle that is fairly easy to conceptualize.

Justification for teaching a functional language in this course:

• Scheme has a comparatively simple computational model but is as powerful as any other programming language.
• What you learn to program in Scheme will make learning other programming languages easier.

Scheme is a functional programming language. Scheme programmers design functions for solving problems that are much like expressions used in mathematics.

Just like in math class, a function takes 0 or more inputs, generates a single output, and causes possibly one or more side effects (drawings of scheme functions with and without side effects.

A non-destructive function has either no side effects or only harmless side effects.

Ch. 2 Expressions vs. Scheme Data

___ Character sequences are used to denote data. A Scheme program is a sequence of characters.
   The syntax rules govern what is and is not a legal Scheme program. The semantic rules tell us which datum each legal expression denotes.
___ Expressions are character sequences. In Scheme, each legal expression denotes a piece of data.
The universe of Scheme is partitioned into data types, and each datum belongs to one and only one data type (go over universe of Scheme picture and partitioning).

Primitive data types: atomic, not composed of smaller parts that a Scheme function can access.

All primitive (built-in types) evaluate to themselves (what you type is what is returned).

Examples of atomic primitive data types include:

- **Numbers**: character sequences such as 2, -57, 22.01, 7/5 are legal Scheme expressions according to syntax rules. Semantics of Scheme tell us these expressions denote numbers.

  If we type the character sequence 2, we say that it denotes \( \rightarrow \) the number two. This is because we do not know or care how the computer represents the number 2, just so it retains the meaning we desire.

  \[
  2 \quad \rightarrow \quad \text{the number two} \\
  -57 \quad \rightarrow \quad \text{the number negative fifty-seven}
  \]

- **Booleans**: the character sequences \#t true \#f false are legal Scheme expressions by the syntax rules of Scheme. The character sequences true and false are aliases for \#t and \#f

  \[
  \#t \quad \rightarrow \quad \text{the true truth value} \\
  \#f \quad \rightarrow \quad \text{the false truth value}
  \]

- **Empty list**: All lists begin with an open parenthesis and end with a close parenthesis. The empty list is the smallest list possible. The character sequences empty and null are aliases for ().

  \[
  () \quad \rightarrow \quad \text{the empty list}
  \]

- **Characters**: represented by the character sequences \#\a, \#\b, \ldots \#\z.

- **Quoted Symbols**: *This data type is not given in our textbook.* The data type called symbol in Prof. Hunsberger’s book is a defined name for a constant or function. I will talk about this type when I cover the define special form. When I refer to symbols, I mean “quoted symbols”. A character sequence that is preceded by an apostrophe and contains no whitespace. Aside: It is not possible to create a quoted symbol out of a number, boolean, character, or string.

  ‘hello \quad \rightarrow \quad \text{the character sequence hello} \\
  ‘Howdy! \quad \rightarrow \quad \text{the character sequence Howdy!} \\
  ‘Hello, world \quad \rightarrow \quad \text{**Error** no whitespace allowed in symbols.}

- **Functions**: represented by the character sequence stored in the GE associated with the name of the function.

  \[
  + \quad \rightarrow \quad \text{The function for addition} \\
  * \quad \rightarrow \quad \text{The function for multiplication} \\
  abs \quad \rightarrow \quad \text{The function for absolute value}
  \]

- **Void**: Used to represent “no value”. There is no character sequence that denotes void. However, void is a function, which Racket shows as \#\langle\text{procedure: void}\rangle when you enter void in the IW. If you call the void function like this: (void), you get the “no value” return.
• Compound primitive data types: composed of smaller parts that are scheme data types we can access. Examples of compound data types include
  - **Strings:** strings are composed of characters that can be accessed by a Scheme program. They are introduced in Ch. 2 because they are useful in printing functions that you will use early on. Strings are delimited by quotation marks (" ").
    
    "hello" → the string “hello”
    “Howdy!” → the string “Howdy!”
    “Hello, world” → the string “Hello, world”
  
  Note: strings can contain whitespace but symbols cannot.
  
- Other compound data types include vectors, non-empty lists, user-created data types (structs), and images.

**Ch. 3 Evaluating Scheme Data: The eval function**

The one action of a Scheme computer is to **evaluate** Scheme data. Evaluation is done by the computer and involves Scheme data, not character sequences. Evaluation is at the core of the Scheme computational model.

Evaluation is a function called **eval** that takes 0 or more inputs and generates a single output. If the inputs to the evaluation function are valid Scheme data, then the function produces a valid Scheme datum. Usually, application of the **eval** function does not generate any side effects (with important exceptions discussed in later chapters).

The double arrow (⇒) represents the application of the **eval** function to some Scheme data (the input) to generate some, perhaps different, type of Scheme datum (the output). In other words, the output datum is the result of evaluating the input(s) (cf. Figs. 3.1 and 3.2, page 12).

The **eval** function acts like the **identity function** for numbers, booleans, the empty list, characters, quoted symbols, and strings (cf. Fig. 3.2, page 12). That’s why these data types are said to evaluate to themselves.

---

Evaluating unquoted symbols is different from evaluation of primitive types. Symbols are used as names for defined **constants** and **functions**. They are evaluated with respect to an environment. An environment is a context within which Scheme data get evaluated (but an environment is not a data type). You can think of an environment as a table of entries, each entry pairing a defined symbol s with a value v of any Scheme data type.

The **define** special form is the first keyword we will address. This keyword has two arguments. When a “(" is followed by the keyword **define**, the first expression after the **define** keyword is a symbol that is written into the Global Environment (GE), not evaluated. The interpretation of the **define** special form differs from default evaluation of functions because only the last argument is evaluated; **define** returns only void, but has the side effect of writing a new name and value into the GE.
As shown in Fig. 3.3 on page 13 of our textbook, with respect to the environment shown on that page, the symbol $NUM$ evaluates to the number three, $XYZ$ to the number two, $BOOLIE$ to the boolean true, and $STRINGY$ to the string “hello”. These constants are written into the GE using the following invocations of the define special form:

```
(define NUM 3)
(define XYZ 2)
(define BOOLIE #t)
(define STRINGY "hello")
```

The default environment when you open DrRacket is called the Global Environment (GE). After the previous 4 define statements, you can conceptualize the GE as a table that includes names for the constant values 3, 2, #t "hello":

<table>
<thead>
<tr>
<th>NAME</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUM</td>
<td>3</td>
</tr>
<tr>
<td>XYZ</td>
<td>2</td>
</tr>
<tr>
<td>BOOLIE</td>
<td>#t</td>
</tr>
<tr>
<td>STRINGY</td>
<td>&quot;hello&quot;</td>
</tr>
</tbody>
</table>

If a symbol does not have a corresponding entry in the GE, then evaluating that symbol with respect to the GE is undefined and there will be an error when the symbol is used in an expression.

Another data type that can be named in the GE is a function. We will talk about that in the next lecture.

Main points of Chapter 3:

i. Each valid expression—which is a character sequence—denotes a Scheme datum.
ii. Each Scheme datum evaluates to a (possibly different type of) Scheme datum.
iii. It is possible for the evaluation to result in an error.
iv. Symbols (constant and function names) are evaluated by looking up the name in the closest containing environment and returning the value associated with that name.