Plan for 101 Day 1 and 2:  
Introduction to Problem Solving and Abstraction  
September 4 and 9 2019

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

__ Start by reading syllabus passed out in a handout. Ask: who already knows any programming language/taken any programming courses.

__ 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. While this is a rather slow approach, it usually leads to faster results because it helps to avoid common mistakes that can frustrate beginning programmers.

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 because:

  • 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.
  • 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—Give side-effect examples).
  • 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. 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 (→) 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 → the number *two*
- -57 → 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 → the true truth value
- #f → 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 ()

- () → the empty list

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

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

- + → The function for *addition*
- * → The function for *multiplication*
- abs → 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 #\(procedure: void\) 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 parts, called 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.

Literal values for numbers, booleans, the empty list, symbols and strings are given at the end of Ch. 2. There is no literal value for the void data type.

---

1We will not usually use complex syntax rules, e.g. scientific notation or inexact numbers for numerical expressions.
Ch. 3 Evaluating Scheme Data: The eval function – Start 9/9/19

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 evaluation (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 evaluation function acts like the identity function for numbers, booleans, the empty list, characters, quoted symbols, and strings (cf. Fig. 3.2, page 12).

Evaluating symbols is different from evaluation of other primitive types.

Symbols are used as names for variables. 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.

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”.

The default environment when you open DrRacket is called the Global Environment (GE).

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

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 (variable and function names) are evaluated by looking up the name in the closest containing environment and returning the value associated with that name.