Notes on Abelson and Sussman, *Structure and Interpretation of Computer Programs*

Chapter 1, "Building Abstractions with Procedures"

Cameron Clarke

July 2019

---

1. Building Abstractions with Procedures (pp. 1-106)

- Target of study: **computational processes** (1)
  - intuition: "Computational processes are abstract beings that inhabit computers" (1)
- Processes manipulate **data** (1)
- Evolution of a process is directed by a pattern of rules called a **program** (2)
- An upshot: "**People create programs to direct processes**" (2)
- Programs are composed from **programming languages** (2)
- A framing thought, among others (2-3)

  > Well-designed computational systems, like well-designed automobiles or nuclear reactors, are designed in a modular manner, so that parts can be constructed, replaced, and debugged separately.

Programming in Lisp

- "[...] our **procedural** thoughts will be expressed in Lisp" (3)
- Lisp **interpreter**: "a machine that carries out the processes described in the Lisp language" (3)
- Interesting historical note, about one of Lisp's original use cases (4)

  > Lisp [...] was designed to provide symbol-manipulating capabilities for attacking programming problems such as the symbolic differentiation and integration of algebraic expressions.

- A note about dialects of Lisp (4)

  > Lisp is by now a family of dialects, which, while sharing most of the original features, may differ from one another in significant ways

  - This book makes use of the Scheme dialect of Lisp
- A main reason why Lisp is being used as the framework for the book's discussion of programming (5)

> If Lisp is not a mainstream language, why are we using it as the framework for our discussion of programming? Because the language possesses unique features that make it an excellent medium for studying important programming constructs and data structures for relating them to the linguistic features that support them. **The most significant of these features is the fact that Lisp descriptions of processes, called *procedures*, can themselves be represented and manipulated as Lisp data**. The importance of this is that there are powerful program-design techniques that rely on the ability to **blur the traditional distinction between "passive" data and "active" processes**.

  - The above facts make Lisp good at writing programs that **manipulate other programs as data** (like interpreters and compilers)

1.1 The Elements of Programming (pp. 6-39)

- A powerful programming language can be used as: (6)
  - a means for instructing a computer to perform tasks
  - a framework within which we organize our ideas about processes
- Combining simple ideas into complex ideas (6)

  > [...] when we describe a language, we should **pay particular attention to the means that the language provides for combining simple ideas to form more complex ideas**.

  - Three mechanisms for doing this
    1. **Primitive expressions**, which represent the simplest entities the language is concerned with
    2. **Means of combination**, by which compound elements are built from simpler ones
    3. **Means of abstraction**, by which compound elements can be named and manipulated as units
- Two kinds of elements in programming: **procedures** and **data** (6)
  - Data is the "stuff" that we manipulate; procedures are the rules for manipulating the data
- An upshot: (6)

  > [...] any powerful programming language should be able to describe primitive data and primitive procedures and should have methods for combining and abstracting procedures and data.

1.1.1 Expressions (pp. 7-10)

- An initial framing (7)

  > Imagine that you are sitting at a computer terminal. You type an *expression*, and the interpreter responds by displaying the result of its *evaluating* that expression.

- Some vocab (8)
  - **combinations**: expressions formed by delimiting a list of expressions within parentheses, in order to denote procedure application
  - **operator**: leftmost element in the list
  - **operand**: other elements in the list than the operator
  - **argument**: "The value of a combination is obtained by applying the procedure specified by the operator to the *arguments* that are the values of the operands"
- Scheme uses **prefix notation**: operator to the left of the operands (8)
  - Advantages of prefix notation (8-9)
    * can accommodate procedures that may take an arbitrary number of arguments
      · e.g., `(+ 21 35 12 7 5 6)`, which evaluates to `86`
    * extends in a straightforward way to allow combinations to be *nested*, i.e., combinations themselves can have combinations as elements
      · e.g., `(+ (* 3 5) (- 10 6))`, which evaluates to `19`
      · might think to use a *pretty-printing* convention—where each long combination is written so that the operands are aligned vertically—if you have lots of nested things, like

        ```
        (+ (* 3
              (+ (* 2 4)
                 (+ 3 5)))
           (+ (- 10 7)
              6))
        ```

        instead of

        ```
        (+ (* 3 (+ (* 2 4) (+ 3 5))) (+ (- 10 7) 6))
        ```
- Scheme interpreter runs in a **REPL** (*read-eval-print loop*)
  - Interesting note here (10)

    > Observe in particular that it is not necessary to explicitly instruct the interpreter to print the value of the expression.

1.1.2 Naming and the Environment (pp. 10-12)

- [TODO]

1.1.3 Evaluating Combinations (pp. 12-15)

- [TODO]

1.1.4 Compound Procedures (pp. 15-18)

- [TODO]

1.1.5 The Substitution Model for Procedure Application (pp. 18-22)

- [TODO]

1.1.6 Conditional Expressions and Predicates (pp. 22-27)

- [TODO]

1.1.7 Example: Square Roots by Newton's Method (pp. 27-33)

- [TODO]

1.1.8 Procedures as Black-Box Abstractions (pp. 33-39)

- [TODO]

1.2 Procedures and the Processes They Generate (pp. 40-74)

1.3 Formulating Abstractions with Higher-Order Procedures (pp. 74-106)