# SICP notes

Mark Tropin

September-October 2023

**DISCLAIMER: This is not a summary of Structure and Interpretation of Computer Programs [SICP]. These are my personal notes, i.e. non-obvious things mentioned in the book that I find interesting.**

## Chapter 1.

To evaluate a combination,

- evaluate the operator
- evaluate the operands
- apply the operator to the operands

It is important to note that:

- EVALUATE is recursive.
  - We evaluate the operator and the operands using the same logic.
- EVALUATE means "reduce to a value".
  - Operators are reduced to built-in instructions ($\oplus$, $\otimes$, ...) that correspond to machine instructions (CPU instruction set). Operands are reduced to numeric values (by applying operators to combinations or looking up their value in the environment).
    $\Longrightarrow$ "value" means primitive. "value" means irreducible.
    No more evaluation rules apply. See theorems 3.5.7, 3.5.8 in [TAPL].

p.12, footnote 14:

This is how `let` works. `Let` basically adds a new symbol to the *local* environment. It is used in evaluating the final expression in a block of code. This value (because it is local) is then destroyed.

End of section 1.1.4: You can't tell a user-defined compound procedure from a primitive. That is the function of the environment: to make user-defined definitions usable like primitives. It is essentially abstracting away definitions by *pointing* to them via names.

### Evaluation strategies.

Section 1.1.5 defines normal-order evaluation and applicative-order evaluation.
In Scala we have *call-by-name* and *call-by-value*.

- call-by-value $\Longleftrightarrow$ applicative-order evaluation
- call-by-name $\Longleftrightarrow$ normal-order evaluation
  - applicative-order evaluation $\Rightarrow$ "evaluate the arguments and then apply"
  - normal-order evaluation $\Rightarrow$ "fully expand and then reduce"

### Bound variables and free variables

TODO section 1.1.8

### Linear recursion and iteration

TODO section 1.2.1

If the iterative process is interrupted, we can resume computation given the last recorded state. This is impossible for recursion. Iteration does not require the use of auxiliary memory, recursive processes require a stack.

| Type of process | Time complexity | Space complexity |
|---|---|---|
| Linear recursive process | $\mathcal{O}(n)$ | $\mathcal{O}(n)$ |
| Linear iterative process | $\mathcal{O}(n)$ | $\mathcal{O}(1)$ |

Table 1: Recursion & Iteration.

### `Lambda` semantics and `let` semantics

TODO section 1.3.2

### Functions as first-class citizens

TODO p.77, footnote 66

## References


[SICP] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. Second edition. MIT Press, 1996. ISBN: 0-262-01153-0.

[TAPL] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002. ISBN: 0-262-16209-1.