SICP Notes

James Sungarda

May 24, 2025

1 Programming Languages

Programming languages combine simple ideas to complex ideas, they are composed of:

- **Primitive expressions** - the simplest elements of a language, such as *numbers and variables*.
- **Compound expressions** - combinations of primitive expressions that *produce new values*, such as arithmetic operations.
- **Evaluation rules** - the rules that define how expressions are evaluated to produce values.
- **Environment** - the context in which expressions are evaluated, including variable *bindings* and function *definitions*.

2 Evaluating methods

2.1 Substitution Model

The substitution model is a method of evaluating expressions by replacing variables with their values. It involves:

- Identifying the variables in an expression.
- Replacing each variable with its corresponding value.
- Simplifying the resulting expression until a final value is obtained.

2.2 Normal Order Evaluation

Normal order evaluation is a strategy for evaluating expressions where the outermost expressions are evaluated first. It involves:

- Delaying the evaluation of expressions until their values are needed.
- Ensuring that expressions are evaluated in a way that avoids unnecessary computations.
- Allowing for the evaluation of expressions that may not terminate under other evaluation strategies. (See 2.2.1)

Key idea in this strategy is that it evaluates arguments *only when they are needed*, which can lead to more efficient computations in some cases.

2.2.1 Implications of each strategy

Consider the following program

```
function p() { return p(); }

function test(x, y) {
    return x === 0 ? 0 : y;
}
```

In this example, the function `p` will lead to an infinite loop if evaluated, while the function `test` will return 0 if `x` is 0, otherwise it returns `y`.
The implications of evaluation strategies can lead to different outcomes based on how expressions are evaluated.

- **Substitution Model** would lead to an infinite loop when evaluating `p()`.
- **Normal Order Evaluation** would also lead to an infinite loop for `p()` but would allow `test(0, 5)` to return 0 without evaluating `y`.

3 Conditional Expressions

Consider the following expression:

```
p ? x : y
```

- `p` is a **predicate** that evaluates to either true or false.
- `x` is the **consequent** that is evaluated if `p` is true.
- `y` is the **alternative** that is evaluated if `p` is false.

The interpreter starts by evaluating the predicate `p`.
If `p` is true, it evaluates and returns `x`; if `p` is false, it evaluates and returns `y`.

3.1 Predicates

Primitive predicates include $>=$, $>$, $=$, $<=$, and $<$. These predicates are used to compare values and return a boolean result (true or false).

3.1.1 Compound Predicates

Compound predicates are formed by combining primitive predicates using logical operators such as `and`, `or`, and `not`. These operators allow for more complex conditions to be evaluated.

4 Functions as building blocks

Functions are *black boxes* that take inputs and produce outputs, and suppresses the details of how they work. The parameters of a function are local to the body of the function.

4.1 Best practice

Consider a function `sqrt(x)` which uses Newton's method to compute the square root of x:
It contains several helper functions $average(x, y)$ and $good\_enough(guess, x)$. These helper functions should be defined *within* the body of `sqrt(x)`.
The end-user should not need to know about these helper functions.