SICP Course Notes

Tom O'Brien

December 4, 2013

Contents

1 Introduction

> Lisp is worth learning for the profound enlightenment experience you will have when you finally get it; that experience will make you a better programmer for the rest of your days, even if you never actually use Lisp itself a lot.
>
> – Eric Raymond, "How to Become a Hacker"

The Structure and Interpretation of Computer Programs is regarded as one of the classic textbooks of Computer Science.

In this pilot course we shall cover material from Chapter 1, as well as learn the basics of the Scheme programming language. Scheme is a dialect of Lisp with few special forms and a simple syntax. In this year 2013 object-oriented programming is (for better or for worse) ubiquitous, though the rise of languages such as Clojure, Scala and Erlang indicate the philosophies of functional programming, having been long-forgotten to all but the most wizened of greybeards, are once again coming back into fashion.

So why Scheme? (It's arguable that) we have come a long way since 1986 when the Wizard Book was last published, so why not adapt the material to a programming language more of our time? There are 3 reasons:

1. Scheme is in fact timeless.
2. Sooner or later you will encounter an exercise requiring a feature of Scheme a replacement langauge may lack. For example, Clojure lacks tail-call optimisation, Scala is awful and no one actually understands Erlang.
3. SICP was written with Scheme in mind, and I believe it would be irreverent to the point of insult to pre-suppose a more suitable language.

These notes are broken-down into an introduction, three chapters on the material, as well as an appendix describing the mathematical background required for some exercises.

2 Lecture 1 - The Elements of Programming

2.1 Basic syntax

Given any new programming language, we must always ask these three questions of it:

1. What are its primitive elements?
2. What are its means of combining these atomic units into compound data?
3. What is its means of abstraction - how do we name our compound data, to be treated as if they themselves are primitive data?

The aim of this section is to introduce the Scheme programming language and make a first attempt at answering these questions.

Scheme is a dialect of Lisp. Its primitive data types are numbers, booleans, characters and symbols, though we restrict ourselves for the moment to numbers (and the occasional boolean). To add two numbers - a combination of two primitive data types - in Scheme, we write:

```
(+ 1 1)
```

Scheme programs are built out of **symbolic-expressions** of the form

```
(<operator> <operand-1> <operand-2> ... <operand-n>)
```

Operands may themselves be combinations, giving us expressions such as

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

There are two things we can note about the Scheme syntax:

1. An expression always begins with an operation, followed by some number of operands. While potentially confusing at first, this prefix notation is useful as it allows for potentially arbitrarily many operands to be given to a function; in a C-style language, what we might write as

   ```
   1 + 2 + 3 + 4
   ```

   we can write in Scheme as

   ```
   (+ 1 2 3 4)
   ```

2. There are a lot of parentheses. This is perhaps the largest psychological hurdle one must overcome when first meeting Lisp syntax. They have the benefit of total disambiguation in terms of what is calculated when; there is no such thing as a table of operator precedence in Lisp. Many editors with modes for dealing with Lisp will aid in navigating and balancing parentheses. To witness an Emacs master glide over and rebalance her parentheses with total mastery is an awesome sight.

We assign names to data with `define`:

```
(define pi 3.14)

(define radius 5)

(define area-of-circle-of-radius-five
  (* pi radius radius))
```

So, we can name individual pieces of data, but that is of little use unless we can name processes for dealing with data:

```
(define (square x)
  (* x x))

(square 4) ; 16

(square 10) ; 100
```

We can define functions another way, using the `lambda` construction:

```
(define square
  (lambda (x)
    (* x x)))
```

The expression `(lambda (x) (* x x))` means:

> The function of one argument x which returns x multiplied by itself. That is, the function $x \mapsto x^2$.

2.2 A model for evaluation

Almost all Scheme functions will obey the following rule for evaluation. To evaluate a Scheme expression:

1. Evaluate each subexpression.
2. Apply the procedure that is the value of the leftmost subexpression to the evaluated subexpressions.

This rule is recursively defined, with the understanding that primitive elements like numbers and built-in functions evaluate to themselves.

Any function which does not follow this model for evaluation is called a special form. Each special form has its own model for evaluation, but there are only a handful to remember.

Examples of special forms already encountered are `define` and `lambda` (why?).

2.3 Case analysis

Any half-decent programming language must be able to ask questions of data, and be able to branch accordingly. In Scheme, we ask questions with `cond`:

```
(define abs
  (lambda (x)
    (cond
     ((< x 0) (- x))
     ((= x 0) 0)
     ((> x 0) x))))
```

Another special form, `cond` takes an arbitrary number of conditional clauses of the form:

```
(<predicate> <consequent-expression>)
```

evaluating each in-turn. When `cond` encounters the first predicate that returns `#t` (true), its corresponding consequent expression is evaluated and returned. If no provided predicates return true, `cond` does not return a value. A default case can be provided with `else`:

```
(define abs
  (lambda (x)
    (cond
     ((< x 0) (- x))
     ((= x 0) 0)
     (else x))))
```

The second conditional clause in the above example is superfluous, so we could instead write:

```
(define abs
  (lambda (x)
    (cond
     ((< x 0) (- x))
     (else x))))
```

In this case, we are in a traditional **if-else** scenario, so we can use Scheme's built-in `if` special form:

```
(define abs
  (lambda (x)
    (if (< x 0)
(- x)
x)))
```

The form `if` takes precisely three arguments: a predicate, an if-clause to be evaluated if the predicate evaluates to `#t`, and an else clause to be evaluated otherwise. Note that the contional clauses that comprise `cond` and `if` are not evaluated until necessary, so we are free to write functions like:

```
(define naughty-function
  (lambda (x)
    (if (< x 0)
(- x)
(/ 1 0))))

(naughty-function -1) ; 1
(naughty-function 1) ; Division by zero signalled by /
```