Notes on SICP

λ

Bernardo Meurer

November 2017

Preface

During the Spring of 2017 I began exploring some alternatives to C as my main programming language. As much as I like C, doing the things I am interested in took far too long in it, and required a lot more energy than I wanted to give to my hobby projects. Namely, I am interested in arbitrary precision arithmetics and multithreading, specially in the context of hyper-parallelism with technologies such as OpenCL and CUDA. Rust came up as a nice alternative, and since it also struck a chord with me due to being usable for writing kernels[1] as well, I decided to give it a shot. I build hemoglobin with my friend Daniel Sank, and then shortly after decided to try implementing the Prime Swing. I was having trouble implementing the algorithm in Rust in the way I wanted (with generics and heavy polymorphism), so I emailed Peter for a commented listing of the code, and he kindly responded to me with a version of the code in Julia. From there a mysterious turn of events which I no longer recall made me conclude I had to learn a Lisp dialect, and so I bought a book on it and got started.

These are notes taken during my reading of "Structure and Interpretation of Computer Programs" (SICP), my book of choice for learning Scheme. I hope that they are useful to anyone who sets out to finish the book. Keep in note that this paper, together with all code contained in it, is Free Software licensed under the GPLv3.

Setting up

I want to add a small note with regards to the set up I am using to run my Lisp code. I am running an up-to-date version of Arch Linux with the (currently) mainline kernel (4.14). I installed a group of Scheme interpreters and compilers, and in the end I've concluded that one should use `guile` or `racket` as their REPL interpreter. Instructions on how to install either should be available for your platform.

Regarding text editors, I am currently using Visual Studio Code for both Scheme and LaTeX, although I believe most people would argue that Emacs is the "one true editor" when it comes to Lisp. It's up to you, I have found that VSCode works quite well, and is much faster than Atom at the time of writing, while being FOSS as opposed to Sublime Text 3.

[1] https://www.cs.virginia.edu/~bjc8c/papers/levy17rustkernel.pdf

1 Building Abstractions with Procedures

In a Lisp interpreter, much like in Python if you've tried it, simply giving a statement will print it's result on the screen. For example

```
> 2
2
```

Here `>` is showing a "command" given to the interpreter, and naturally issuing `2`, evaluates to (and prints) `2`. One interesting thing about Lisp (and I suppose functional languages in general) is that expressions are given in *prefix form*, meaning that the operator comes fist, followed by all it's operands (sort of like a function).

```
> (+ 1 2 3 4)
10
> (- 1 2 3 4)
-8
> (* 1 2 3 4)
24
> (/ 1 2 3 4)
1/24
```

If you want to define a variable or a function, the behavior is very similar

```
> (define a 10)
> a
10
> (define (square x) (* x x))
> (square a)
100
> (define b (square a))
> b
100
> (square b)
10000
> (define (sum-of-squares x y)
(+ (square x) (square y)))
>  (sum-of-squares 3 4)
25
```

Here `sum-of-squares` is called a *compound procedure*, since it's composed of multiple procedures (duh). If you want to think about compound procedures (to try and understand what's going on) you can use a technique called *substitution model*, where we go substituting the variables by literal values. So suppose we wanted to analyze `(sum-of-squares (* 10 5) (* 2 4))`. Firstly we resolve `(* 10 5)` to `50` and `(* 2 4)` to `8`. Then we resolve `(sum-of-squares 50)` to (+ (square 50) (square 8)), which in turn yields (+ (* 50 50) (* 8 8)), and subsequently `(+ 2500 64)`, and finally `2564`. This is *not* a model for how the actual interpreter works, but it's just to help us have a model of what is happening.

In Lisp conditional statements are called `cond`, and their syntax is as follows

```
> (define (abs x)
(cond ((> x 0) x)
      ((= x 0) 0)
      ((< x 0) (- x))))
> (abs -8)
8
> (abs 8)
8
> (abs 0)
0
```

A generalized view of these conditions is as a set of *clauses*, each containing a *predicate* and a *consequent*.

$$\begin{array}{ll}(\text{cond} & (p_1 e_1) \\ & (p_2 e_2) \\ & \vdots \\ & (p_n e_n))\end{array}$$

Here, p is the predicate, and e the consequent. Compare that model to the concrete example above and see that you identify predicates and consequents. We'll call *predicates* both procedures and expression that return true or false, so `<`, `>`, `=` are all *primitive predicates*. Here are a couple other ways to write that `abs` procedure

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

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

The latter example uses a special conditional `if`, which can only be used when there are exactly two cases. It has general form

$$(\text{if} \quad p \quad e \quad a)$$

Where once again p is the predicate, e is the consequent, and the new symbol a is the *alternative*.

Logical composition operators such as `and`, `or`, `not` allow us to construct compound predicates.

```
> (define x 8)
> (or (> x 5) (< x 10))
#t
> (and (> x 5) (< x 10))
#t
> (not (or (> x 5) (< x 10)))
#f
```