Chapter 1: Building Abstractions with Procedures

Caesar X Insanium

December 17, 2023

Contents

Here are the relevant lectures related to this chapter

- Lecture 1A

- Lecture 1B

- Lecture 2A

A process is an idea of a set of events occurring under the direction of a program.

Lips was invited as a way to express computational thought in a manner using recursion equations. Lisp is an interpreted language that is akin to a mathematical notation. While it used to be slow it became able to translate directly to machine code in order to get that boost in performance. It can now be used in production and as an extension language. In lisp, data and function can be both be returned from function calls.

1 1.1 Elements of Programming

Programming languages can be used to express ideas about instructions that we want a computer to execute as well as about processes and incorporate decision-making. This is composed to 3 things

- primitive expressions: simplest entities and data
- combination: methods to build larger things from smaller ones
- abstractions: ways to reduce complexity by finding implementation details and organizing data and procedures

1.1 1.1.1 Expressions

Expressions are evaluated and interpreter will return a value, usually itself.

Here is an example in lisp

```
89 ;; will return 89 as an integers
(+ 9 10) ;; will evaluate result of adding 9 and 10 and then return
(+ 1 1 1 1) ;; unlimited number of operands
(+ (* 3 8) (- 10 4)) ;; nested expressions
```

Operators and operands are organized into prefix notation where the operator is listed first and operands come afterwards. This hold true for function calls. More complicated expressions are difficult for humans to understand, but an interpreter can easily handle complexity.

Of course such expressions can also be rewritten in a way that would allow for easier human understanding by aligning operators.

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

1.2 1.1.2 Naming and Environment

In lisp things can be named with `define`. This allows easy reference with a value without spelling it out. Assigned variable names are evaluated with their referred value.

```
(define size 2)
(+ size 2) ;; evaluates to 2
```

1.3 1.1.3 Evaluating Combinations

In order to evaluate lisp expression

- evaluate sub-expressions
- apply procedure on data
- return value

Imagine a tree structure that with each expression, there is an operator and a list of values. They can either be literals or expressions that themselves need to be evaluated.

1.4 1.1.4 Compound Procedures

Abstraction technique that allows for a set of instructions to be saved and used multiple times in a program.

Here is procedure that square a number.

```
(define (square x) (* x x))
;; here is example usage
(square 21)
(square (+ 6 7))
(square (square 5))
```

Define a procedure where we multiply a thing by itself. Here we use the `define` keyword to name a procedure, specify that it takes only one parameter and then following expression is that happens to data that is passed.

```
(define (<name> <formal parameters>) (<body expression>))
```

1.5 1.1.5 Substitution Model Procedure Application

Evaluation begins from the top down, in order to begin to evaluate the top most expression, first the bottom expressions must be evaluated recursively. One way is to replace every instance where we call a procedure with the body of said procedure.

One can simply think back to algebra to when we were solving systems of equations. This is known as the substitution model.

Full implementation of interpreter and compiler will be done in this book.

Normal order evaluation is done by humans where full meaning of expression is expanded to using only the most basic ideas and then evaluate. Interpreter use application order evaluation where expressions are evaluated before applying.

1.6 1.1.6 Conditional Expression and Predicates

Method for executing different based on results of tests if required for Turing completeness and is basis for any for decision-making. Here are some examples.

```
(define (abs x)
  (cond ((> x 0) x) ;; this function is very similar to switch statement from C
        ((= x 0) 0)
        ((< x 0) (-x))))
(define (abs x)
  (cond ((< x 0) (-x))
        (else x)))
(define (abs x)
  (if (< x 0)
      (- x)
      x))
```

Here a condition is defined followed by a list of clauses or tests that can be performed.

We see the base form for a conditional statement in lisp.

```
(if <predicate> <consequent> <alternative>)
```

1.6.1 Example: Square Root using Newton's Method

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

(define (improve guess x)
  (average guess (/ x guess)))

(define (average x y)
  (/ (+ x y) 2))

(define (good-enough? guess x)
    (< (abs (- (square guess) x)) 0.001))

(define (sqrt-iter guess x)
  (if (good-enough? guess x)
    guess
    (sqrt-iter (improve guess x)
               x)))

(define (sqrt x)
  (sqrt-iter 1.0 x))
```

1.7 1.1.8 Procedures as Black-Box Abstractions

Procedure Abstraction should allow for encapsulation of lower level procedure and joining them together in order to make larger procedures. Naming arguments generally does not matter to user of a procedure. All that matters is that the function is correct and returns needed value.

Lisp also allows defining private procedures inside other procedures in order to be able to maintain a specific procedure that is important to working of public facing function.

```
(define (public x)
  (define (private y)
    (write y)
    (newline))
  (private x)
```

2 1.2 Procedures and Processes They Generate

In order to make the best use of procedure it is not enough to know how they work one must also know tactics and strategies for using them. In order to get the bast idea on how to design and build a system one must have a good idea on what we want the end result to be. It makes it necessary to plan out much of what we want to do. Procedures are local evolutions of computational processes and as such can be built on top of each other in order to create the bigger result. They are some "shapes" that procedure definitions can follow.

Lisp allows reasoning about and build procedures as if they were mathematical expressions.

2.1 1.2.1 Linear Recursion

The definition of factorial is as follows

$$n! = n * (n-1) * (n-2)...3 * 2 * 1 \tag{1}$$

From this it is logical to assume that n factorial is equal to n times n minus one factorial.

$$n! = n * (n-1)! \tag{2}$$

Here is a more recursive method

```
(define (factorial n)
  (if (= n 1)
    1
    (* n (factorial (- n -1)))
  )
)
```

Here is another method of defining the factorial function.

```
(define (factorial n)
  (define (iter product counter)
    (if (> counter n)
      product
      (iter (* counter product)
            (+ counter 1))))
  (iter 1 1))
```

Expanding expression allows one to see the true "shape" of a procedure as it evaluates. Some have a diamond shape as the expression expands to the simplest term and then contracts as each term is evaluated.

The expansion evaluation of a procedure is known as `deferred operations` in which an is like as the program is being run this is the part where each function is being initial called. And then after that the evaluation of the value is the actual recursion.

With the iterative function definition each time the recursive function is called it is being immediate evaluated before it can be called again. This is `iterative` processes. In this method the state can be tracked with certain variables keeping track of when the process should end if at all. For is a for loop in most languages.

Iterative procedures allow for easy description of the entire state of the process at any given point. Recursive not so much since the compiler/interpreter hide away much of the inner working up to the point in which it is difficult to resume from any position.

Recursive procedures are strictly those that call themselves within their own definition. Iterative procedures hold their state in outside variables.

Most languages define iterative procedures using for/while loop unlike Scheme and any other lisp dialect.

2.2 1.2.2 Tree Recursion

Tree recursion occurs when a procedure calls itself more than once inside its own definition. For example look at this procedure for getting Fibonacci numbers.

```
(define (fib n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (fib (- n 1))
                 (fib (- n 2))))))
```

This example is not efficient because work is repeated. Computing the Fibonacci of any sufficiently big number will result with entire branches of work being recalculate in different branches in the execution of the procedure. The time complexity of this function is exponential. Space complexity grow linearly.

Here is iterative version.

```
(define (fib_i n)
```

```
  (fib-iter 1 0 n))

(define (fib-iter a b counter)
  (if (= counter 0)
    b
    (fib-iter (+ a b) a (- ounter 1))))
```

Here the time complexity is linear.

Tree recursion and iteration are tools and should be used when it is the best tool in the current situation.

2.3 1.2.3 Orders of Growth

Order of growth simply describe who quickly the time it takes for a procedure to finish executing given data size n as n reaches infinity. N can be used to describe the size of a number itself, the number of bit required to describe a piece of data, the number of elements in a data structure.

Such method of describing a procedure grows in computation time is inaccurate, but it is useful in describing how efficient an algorithm really is.

2.4 1.2.4 Exponentiation

One way to calculate the exponent of a value is to use the recursive definition of exponentiation.

$$b^n = b * b^{n-1}, b^0 = 1 \tag{3}$$

Here it is in scheme.

```
(define (expt b n)
  (if (= n 0)
    1
    (* b ( expt b (- n 1)))))
```

It is possible to make a faster procedure by simply using a different algorithm. It can be done using the idea that certain value can be reached faster. For example take following expression.

$$b^4 = b^2 * b^2 \tag{4}$$

Using this

$$b^8$$

can be calculated much faster. Here is an implementation.

```
(define (fast-expt b n)
  (cond ((= n 0) 1)
        ((even? n) (square (fast-expt b (/ n 2))))
        (else (* b (fast-expt b (- n 1))))))
```

Given from this implementation, it can be easy to see that the time complexity goes to logarithmic because each exponent jump is bigger the deeper the recursion goes.

2.5 1.2.5 Greatest Common Divisor

Greatest Common Divisor or GCD is defined as the largest integers that divides integers A and B. Meaning that there are integers x and y such that

$$x * GCD = A$$

and

$$y * GCD = B$$

Finding the GCD is simple since one can take the recursive definition.

$$GCD(a, b) = GCD(b, r) \tag{5}$$

Use it to define this function in scheme.

```
(define (gdc a b) (if (= b 0)
                      a
                      (gcd (remainder a b))))
```

We continuously divide a from b recursively until the final GCD is found.

This algorithm is iterative in nature as each recursion is a tail recursion and grows in logarithmic time.

2.5.1 Example: Testing for Primality

There are two main methods for testing if integer n is a prime number or not.

One strategy is to find if it has any factors.

```
;;Naive Method

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

```
(define (smallest-divisor n)
  (find-divisor n 2))

(define (divides? a b) (= (remainder b a) 0))

(define (find-divisor n test-divisor)
  (cond ((> (square test-divisor) n ) n)
        ((divides? test-divisor n) test-divisor)
        (else (find-divisor n (+ test-divisor 1)))))

(define (prime? n) (= n (smallest-divisor n)))
```

What this implementation does it check all numbers less than square root of n if they are the prime numbers. The logic goes if none of those work then the number n is prime.

The second big strategy is based on Fermat's Little theorem which states basically the idea that is n is a prime number and a is a positive integer less than n then a raised to the n power is congruent to a modulo n.

Two numbers are congruent if when divide by the same number have the same remainder.

```
;; Fermats Theorem applied

(define (expmod base exp m)
  (cond ((= exp 0)
         1)
        ((even? exp)
         (remainder (square (expmod base (/ exp 2) m)) m))
        (else
          (remainder (* base (expmod base (- exp 1)m))m))))
```

This procedure grows logarithmic in accordance to size of input. Using it we can implement the Fermat test.

```
(define (fermat-test n)
  (define (try-it a)
    (= (expmod a n n) a))
  (try-it (+ 1 (random (- n 1)))))
```

The final form. Will take a number and number of attempts to see if it is prime.

```
(define (fast-prime? n times)
  (cond ((= times 0) 1)
        ((fermat-test n) (fast-prime? n (- times 1))) (else 0)
```

However, it has been proven that even if the procedure says that a number passes the test then it does not mean that it is prime. There is also the fact that some non primes that fool the Fermat test for why modulo congruent with a lot of numbers. This algorithm is an example of a `Probabilistic Algorithm` in which there is a chance of error that the algorithm yields the incorrect result.

2.6 Formulating Abstractions with Higher Order Procedures 1.3

The ability to write procedures and function allows for the ability to create program that can work on higher and higher levels of abstraction and reuse instructions and operations without repeat the definitions.

Procedures that only work on numbers can be limiting. `Higher Order` procedures are those that accept functions as arguments and can return functions as results.

2.7 1.3.1 Procedures as Argument

Here we have three procedures.

```
;; Sums Integers from A to B
(define (sum-integers a b)
        (if (> a b)
            0
            (+ a (sum-integers (+ a 1) b))))

;; Computers Sum of Cubes of Integers
(define (sum-cubes a b)
    (if (> a b)
        0
        (+ (cube a) (sum-cubes (+ a 1) b))))

;; Computes Series of multiplied cubes
;;; 1 / (1 * 3) + 1 / (5 * 7) +1 / (9 * 11)
(define (pi-sum a b)
    (if (> a b)
```

```
        0
        (+ (/ 1.0 (* a (+ a 2))) (pi-sum (+ a 4) b))))
```

These are similar procedures, that have the same template and could be automatically generated. Sigma notation is used in order to express how express a method to add the results of a function given a range of integer values.

$$\sum_{n=a}^{b}(f(n) = f(a) + ... + f(b)) \tag{6}$$

The realization that the pattern being emulated is in fact a mathematical summation allows for easy redefinition using the scheme language.

```
(define (sum term a next b)
        (if (> a b)
            0
            (+ (term a)
                (sum term (next a) next b))))

;; term is a function that determines the selection of items being summed
;; next is a function that determines which is the next items after the previus one
    ;; for function term
```

Using some other helper functions it is possible to redefine all the functions.

```
;; redefinition of a previous function
(define (sum-cubex a b)
        (sum cube a inc b))

(define (sum-integers a b)
        (sum identity a inc b))

(define (pi-sum a b)
        (define (pi-term x)
                (/ 1.0 (* x (+ x 2))))
        (define (pi-next x)
                (+ x 4))
        (sum pi-term a pi-next b))
```

Using this same higher order procecure it is possible to define a function to approximate the integral of a function.

```
(define (integral f a b dx)
        (define (add-dx x)
                (+ x dx))
        (* (sum f (+ a (/ dx 2.0)) add-dx b)
           dx))
;; can be used to approximate the integral from 0 to 1 of x^3
(integral cube 0 1 0.01)
;; .24998750000000042 result
(integral cube 0 1 0.001)
;; .249999875000001 result
```

2.8 1.3.2 Constructing Procedures Using Lambda

Scheme allows for methods of defining simple single use functions without giving them names. These are `lambda`, anonymous function that are a definition of a function that does one simple thing and developer forgets about them.

```
(lambda (x) (+ x 4))
;; Format
(lambda (<name> <formal-parameters>) <body>)
```

Usually these are passed to other functions in order to generate higher level procedures and are the hallmark of a functional programming language. An example using previous mentions.

```
(define (pi-sum a b)
        (sum (lambda (x) (/ 1.0 (* x (+ x 2))))
             a
             (lambda (x) (+ x 4))
             b))

(define (integral f a b dx)
        (* (sum f
                (+ a (/ dx 2.0))
                (lambda (x) (+ x dx))
                b)
           dx))
```

The usualy method for defining named function can be thought of as syntactic sugar for lambda.

```
(define (f x) (+ x 1)) ;; is the same as
(define f (lambda (x) (+ x 1)))
```

2.8.1 Using `let` to create local variables

Using the `let` keyword is useful in defining variables with limited scope in order to pollute namespace. Taking the mathematical expression

$$f(x,y) = x(1+(x*y))^2 + 1y(1-y) + (1+(x*y))(1-y) \tag{7}$$

can be simplified to

$$a = 1+(x*y), b = 1-y, f(x,y) = xa^2 + (y*b) + ab \tag{8}$$

Writing scheme code in order to emulate this function would require not only the parameters but also defining the local variables a and b. Here is some normal scheme code.

```
(define (f x y)
        (define (f-helper a b)
                (+ (* x (square a))
                   (* y b)
                   (* a b)))
        (f-helper (+ 1(* x y))
                  (- 1 y)))
```

Here is the simplification. Side note, getting the indentation correct was a pain in the ass.

```
(define (f x y)
    (let ((a (+ 1 (* x y)))
          (b (- 1 y)))

         (+ (* x (square a))
            (* y b)
            (* a b))))
```

General form for let expression is this

```
(let ((<var1> <exp1>)
      (<var2> <exp2>)
      ...
      (<varn> <expn>))
    <body>)
```

Each section can be thought of as its own little mini section with define pairs. The different is that the variables here have limited scope and exist only within the confines of the S expression of let. Even this expression can be thought of as syntactic sugar for a lambda expression that take in certain parameters that are the names of parameters and outputs a list with each item being corresponding to the value of expression.

If a variable already has a value, then the `let` expression overrides it within the scope of the expression, ignoring outside values.

```
(define x 5)
(+ (let ((x 3))
        (+ x (* x 10)))
   x) ;; evaluates to 38
```

Best practices dictates to use `let` in defining variables and `define` in defining internal procedures.

2.9 1.3.3 Procedures as General Methods

Strategies have been introduced that allow for expression numerical methods as scheme procedures, and abstracting away general strategies to be independent of numbers and or even computation involved.

2.9.1 Finding Roots of Equations by Half Interval Methods

Half interval method involved in finding roots of function which average in range of values is found and if prove that a zero exists in interval, the interval is cut in half until the root is found.

The number of steps required to find the root grows logarithmic in relation to size of starting interval. Here is example from book.

```
(define (search f neg-point pos-point)
  (let ((midpoint (average neg-point pos-point)))

       (if (close-enough? neg-point pos-point)
```

```
midpoint
(let ((test-value (f midpoint)))
     (cond ((positive? test-value)
            (search f neg-point mid-point))
           ((negative? test-value)
            (search f midpoint pos-point))
           (else midpoint))))))
```

What this code does is first calculated the midpoint of the interval. Then it checks to see if the interval is close enough, if it is, it returns the midpoint. If not, then it evaluates the function f at the midpoint. If this new value is positive then it calls itself with new interval from beginning to midpoint. Then if negative the new range is from midpoint to end of range. If the midpoint evaluates to zero then it gets returned.

Search should not be called directly and instead suitable interval should first be found and then `search` should be called.

2.9.2 Fixed Points of Functions

Fixed point in function are where the output of a function is the same as the input. These points can be found by applying this calculation. Where x is a guess and applying the same transformation.

$$f(x), f(f(x)), f(f(f(x))), f(f(f(f(x)))) \tag{9}$$

Using this definition it is possible to define procedure that finds a fixed point given a function and initial guess. Here is the example the book gives.

```
(define tolerance 0.00001)

(define (close-enough? x y)
  (< (abs (- x y)) tolerance))
(define (fixed-point f first-guess)
  (define (try guess)
    (let ((next (f guess)))
         (if (close-enough? guess next)
             next
             (try next))))
  (try first-guess))

(fixed-point cos 1.0)
```

This code basically just guesses the fixed point until it gets it close enough based on some criteria.

It can also find the solution to equations. Give for example.

$$y = cos(y) + sin(y) \tag{10}$$

Can be solved with.

```
(fixed-point (lambda (x) (+ (sin x) (cos x))) 1.0)
```

Average Damping can also be used in order to more carefully converged towards the wanted value without getting stuck in an infinite loop.

2.10 1.3.4 Procedures as Returned Values

More power and expression ability can be achieved by using procedures that generate and return procedures. For example, procedure can be defined that can average damp the result of a function.

```
(define (average-damp f)
  (lambda (x) (average x (f x))))
```

What this function does is it take a function and return a new function that takes the average of an input and the result of calling that function on an input. This concept can be used in order to redefine the square root function in terms of the previously define `fixed-point` function.

```
(define (sqrt x)
  (fixed-point (average-damp (lambda (y) (/ x y))) 1.0))

;; Cube Root can also be defined
```

Recognizing patterns that repeat in procedures is an important skill to have as it for more easy reuse of different components.

2.10.1 Newton's Method

Early implementation of square root function involved Newton's method in which solution can be approximated using the identity.

$$f(x) = x - \frac{g(x)}{Dg(x)} \tag{11}$$

The does here is to find x such that f(x) = 0. The bottom term is the derivative of the function g at x. This method does not always converge but does converge enough times such that it is useful approximation.

In order to use this method, the derivative must found, and used. It is defined as

$$g'(x) = \frac{g(x + dx) - g(x)}{dx} \tag{12}$$

The code here is simply returning a new function that is in terms of the passed and uses the definition of the derivative in order to calculate the derivative at a certain value of x.

```
(define dx 0.000001)
(define (deriv g)
  (lambda (x)
        (/ (- (g (+ x dx))
              (g x))
           dx)))
```

A new procedure can be defined that take a function f and returns it's derivative in terms of f. Then Newton's Method can be implemented by using it as a fixed point function.

```
(define (newton-transform g)
  (lambda (x)
          (- (/ (g x)
                ((derive g) x)))))

(define (newtons-method g guess)
  (fixed-point (newton-transform g) guess))
```

The square root function can be redefined in terms of these new functions.

2.10.2 Abstractions and First Class Procedures

Further abstraction can be achieved by formulating the repeat used of the fixed point function and abstracting its repeated use.

```
(define (fixed-point-of-transform g transform guess)
  (fixed-point (transform g) guess))
```

This procedure takes as arguments function g, a function that a solution found, and an initial guess. What is does is it calls the fixed point function that takes the transform procedure and calls it on g, and the guess. The whole purpose is to define a function that can be solved and to solve it. And this can be used to redefine the square root function.

```
(define (sqrt x)
  (fixed-point-of-transform (lambda (y) (/ x y))
                            average-damp
                            1.0))
```

These higher order procedures allow for easy expression of computing methods using more basic elements that already been defined in our code. Overly abstraction code can also de detrimental but the price of not doing so is worse. The key is thinking in terms of abstraction and reusing them whenever code begins to feel repetitive and to apply them and new contexts.

In computer languages there is a concept of first-class elements. These have properties in computer languages in which there are few restrictions and have some features.

- Can be named as variables
- passed as arguments to functions
- returned as result of procedures
- include in data structure

In lisp and other dialects function are first class citizens, with all the above features. Fast and efficient implementation is difficult, but expression are very powerful.