Dedication

“I hope we don’t become missionaries. Don’t feel as if you’re Bible salesmen. The world has too many of those already. What you know about computing other people will learn. Don’ t feel as if the key to successful computing is only in your hands. What’s in your hands, I think and hope, is intelligence: the ability to see the machine as more than when you were first led up to it, that you can make it more.”

1. Building Abstractions with Procedures

Programming in Lisp

- MacLisp (Moon 1978; Pitman 1983), developed at the MIT Project MAX
- Interlisp (Teitelman 1974), developed at Bolt Beranek and Newman Inc. and the Xerox Palo Alto Research Center.
- Portable Standard Lisp (Hearn 1969; Gross 1982) designed to be easily portable between different machines.
- Scheme (Steel and Sussman 1975), was invented in 1975 by Guy lewis Steele Jr. and Gerald Jay Sussman of the MIT Artificial Intelligence Laboratory and later reimplemented for instructional use at MIT. Scheme became an IEEE standard in 1990 (IEEE 1990).
- Common Lisp (Steele 1982, Steele 1990) was developed by the Lisp community to combine features from the earlier Lisp dialects to make an industrial standard for Lisp. Common Lisp became an ANSI standard in 1994 (ANSI 1994).

MacLisp spawned a number of subdialects, such as Franz Lisp, which was developed at the University of California at Berkeley, and Zetalisp (Moon and Weinreb 1981), which was based on a special-purpose processor designed at the MIT Artificial Intelligence Laboratory to run Lisp very efficiently.

1.1. The Elements of Programming

Every powerful language has three mechanism for accomplishing this:

- Primitive expression
- Means of combination
- Means of abstraction

1.1.1. Expressions

It’s known as prefix notation:

```
(+ 21 35 12 7)
```

There is no limit (in principle) to the depth of such nesting and to the overall complexity of the expressions:

```
(+ (* 3 5) (- 10 6))
```

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

We can help ourselves by writing such an expression in the form:

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

Following a formatting convention known as pretty-printing.

1.1.2. Naming and the Environment

In the Scheme, we name things with define, typing:

```
(define size 2)
```

1.1.3. Evaluating Combinations

1.1.4. Compound Procedures

“To square something, multiply it by itself.” This is expressed in out languages as :

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

The general form of a procedure definition is:

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

1.1.5. The Substitution Model for Procedure Application

1.1.6. Conditional Expressions and Predicates

We define a procedure that taking different actions in the different cases according to the rule, and it's called a case analysis:

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

The general form of a conditional expression is:

```
(cond (<p1> <e1>)
       ...
      (<p2> <e2>))
```

Another way to write the absolute-value procedure is :

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

Here is yet another way to write the absolute-value procedure:

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

1.1.7. Example: Square Roots by Newton's Method

1.1.8. Procedures as Black-Box Abstractions

1.2. Procedures and the Processes They Generate

In this section we will examine some common "shapes" for processes generated by simple procedures.

1.2.1. Linear Recursion and Iteration

In beginning we define the factorial function as:

```
n! = n * (n-1) * (n-2) ... 3 * 2 * 1.
```

One way to compute factorials is to make use of the observation that n! Is equal to n times (n-1)! For any positive integer n:

```
n! = n * [(n-1) * (n-2) ... 3 * 2 * 1] = n * (n-1)!
```

This observation translate directly into a procedure:

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

1.2.2. Tree Recursion

Consider computing the sequence of Fibonacci numbers, in which each number is the sum of the preceding two:

```
0, 1, 1, 2, 3, 5, 8, 12, 21, ...
```

We can translate the definition into a recursive procedure for it:

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

And then, we can compute Fibonacci numbers iteratively using the procedure,it's a linear iteration:

```
(if (= count 0)
    b
    (fib-iter (+ a b) a (- count 1))))
```

1.2.3. Orders of Growth

Similar to complexity theory.

1.2.4. Exponentiation

Consider the problem of computing the exponential of a given number, we can translates the recursive definition into the procedure:

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

And we can readily formulate an equivalent linear iteration:

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

And then, we can take advantage of the formulation:

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

(define (even? n)
  (= (remainder n 2) 0))
```

1.2.5. Greatest Common Divisors

There is a express Euclid’s Algorithm as a procedure:

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

```
    a
    (gcd b (remainder a b))))
```

It's a wonderful algorithm to find common divisors by recurve. The steps like this:

```
GCD(206, 40) = GCD(40, 6)
             = GCD(6, 4)
             = GCD(4, 2)
             = GCD(2, 0)
             = 2
```

1.3. Formulating Abstractions with Higher-Order Procedures

1.3.1. Procedures as Arguments

This section we use procedure as arguments, the full code like this, and due to the racket not having an inc function, I define one.

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

(define (identity x) x)
(define (inc x) (+ x 1))
(define (sum-integers a b)
   (sum identity a inc b))

(sum-integers 1 10)
```

It is work.

1.3.2. Constructing Procedures Using lambda

The procedure is the same as another.

```
(define (plus4 x) (+ x 4))
(define plus4 (lambda (x) (+ x 4)))
```

1.3.3. Procedures as General Methods

1.3.4. Procedures as Returned Values

In general, programming languages impose restrictions on the ways in which computational elements can be manipulated. Elements with the fewest restrictions are said to have first-class status. Some of the “rights and privileges” of first-class elements are:

- They may be named by variables.
- They may be passed as arguments to procedures.
- They may be returned as the results of procedures.
- They may be included in data structures.

2. Building Abstractions with Data

Aaaaaaaaaaaaaaah ….....