DDG Continuing Education

...or Study Group...

...or something that sounds cooler...

Informal Goals

1. Learn new things related to technology.
2. Learn from each other.
3. Foster inter-team building.
4. To become better engineers.

Search for DDG Study Group in Asana.

Structure and Interpretation of Computer Programs *(SICP)*

by Harold Abelson and Gerald Jay Sussman

Brief intro to Lisp

REPL

Read Eval Print Loop

```
faraday:sicp-exercises mas$ scheme

MIT/GNU Scheme running under OS X
Type `^C' (control-C) followed by `H' to obtain information about interrupts.

Copyright (C) 2014 Massachusetts Institute of Technology
This is free software; see the source for copying conditions. There is NO warranty; not
 even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

Image saved on Saturday May 17, 2014 at 2:39:25 AM
Release 9.2 || Microcode 15.3 || Runtime 15.7 || SF 4.41 || LIAR/x86-64 4.118 || Edwin
3.116

1 ]=> (+ 1 1)

;Value: 2

1 ]=>
```

Basically a shell

Prefix Notation

```
1 ]=> (+ (* 3 5) (- 10 6))

;Value: 19
```

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

is the same as:

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

Defining (Binding) Variables

Lisp:

```
(define pi 3.14159)
(define radius 10)

(* pi (* radius radius))
> 314.159

(define circumference (* 2 pi radius))

circumference
> 62.8318
```

Perl:

```
my $pi = 3.14159;
my $radius = 10;

$pi * $radius * $radius;
> 314.159

$circumference = 2 * pi * $radius;

$circumference;
> 62.8318
```

Procedures

Lisp:

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

(square 2)
> 4
```

Perl:

```
sub square {
    my $x = @_;

    return $x * $x;
}

square(2);
> 4
```

Special form:

```
(define (name formal-parameters) body)
```

Conditionals: cond

Lisp:

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

Perl:

```
sub abs {
    my ($x) = @_;

    if ($x > 0) {
        return $x;
    } elsif ($x == 0) {
        return 0;
    } elsif ($x < 0) {
        return - $x;
    }
}
```

Special form:

```
(cond ((predicate1) expression1)
      ((predicate2) expression2)
      ...
      ((predicateN) expressionN)
      (else default-expression))
```

Conditionals: if

Lisp:

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

Perl:

```
sub abs {
    my ($x) = @_;

    if ($x < 0) {
        return $x;
    } else {
        return - $x;
    }
}
```

```
Special form:
```

```
(if (predicate) (then-expression) (else-expression))
```

** if can only have a single expression in then or else.*

Printing!

Lisp:

```
(display foo) (newline)
```

Perl:

```
print "$foo\n";
```

1.1 The Elements of Programming

Every powerful language has three mechanisms for combining simple ideas to form more complex ideas:

1. ***primitive expressions***, which represent the simplest entities the language is concerned with,
2. ***means of combination***, by which compound elements are built from simpler ones, and
3. ***means of abstraction***, by which compound elements can be named and manipulated as units.

The Substitution Model

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

(define (sum-of-squares x y)
  (+ (square x) (square y)))

(define (f a)
  (sum-of-squares (+ a 1) (* a 2)))
```

Evaluation:

```scheme
(f 5)
```

```scheme
(sum-of-squares (+ 5 1) (* 5 2))
```

```scheme
(+ (square 6) (square 10))
```

```scheme
(+ (* 6 6) (* 10 10))
```

```scheme
(+ 36 100)
```

136

Helps us to think about procedure application, this is not how the interpreter actually works.

Applicative Order Evaluation

1. Evaluate the subexpressions of the combination.
2. Apply the procedure that is the value of the leftmost subexpression (the operator) to the arguments that are the values of the other subexpressions (the operands).

Essentially, what we just saw:

```
(f 5)
(sum-of-squares (+ 5 1) (* 5 2))
(+ (square 6) (square 10))
(+ (* 6 6) (* 10 10))
(+ 36 100)
136
```

The Lisp interpreter uses this applicative order evaluation.

Normal Order Evaluation

Doesn't evaluate operands until their values are needed.

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

(define (sum-of-squares x y)
  (+ (square x) (square y)))

(define (f a)
  (sum-of-squares (+ a 1) (* a 2)))
```

```
(f 5)
```

```
(sum-of-squares (+ 5 1) (* 5 2))
```

```
(+ (square (+ 5 1))
   (square (* 5 2)))
```

```
(+ (* (+ 5 1) (+ 5 1))
   (* (* 5 2) (* 5 2)))
```

```
(+ (* 6 6)
   (* 10 10))
```

```
(+ 36 100)
```

Normal vs. Applicative Order Evaluation

```
(define (p) (p))

(define (test x y)
  (if (= x 0)
      0
      y))

(test 0 (p))
```

Exercise 1.5

Under applicative order evaluation

```
(test 0 (p))
```

will expand forever.

Procedures as Black Boxes

```
             sqrt
              |
          sqrt-iter
           /      \
    good-enough   improve
    /        \         \
square       abs      average
```

- Computing square roots breaks up naturally into a number of subproblems.
- Don't break things up arbitrarily.

Scoping

Compare:

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

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

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

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

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

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

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

  (sqrt-iter 1.0))
```

Scoping (cont.)

Lexical scoping defines how variable names are resolved in nested functions: inner functions contain the scope of parent functions.

Wrapping-up

- ***primitive expressions***, which represent the simplest entities the language is concerned with,

  ```
  +, *, <, = or 42, 3.14
  ```

- ***means of combination***, by which compound elements are built from simpler ones, and

  ```
  ( ) composing functions and building combinations with
  operators, also if, cond
  ```

- ***means of abstraction***, by which compound elements can be named and manipulated as units.

  ```
  define
  ```

That's all for section 1.1.

Thanks!