definition var

(define <var name> <value>)

(define pi 3.14)

return: pi

definition procedure(func)

(define (<name> <paras>) <body>)

```
(define (factorial n)
    (if (= n 0)
   1
    (* n (factorial (- n 1)))) )
```

return: procedure name

define list

well-formed list (Linked list)	(cons 1 (cons 2 nil))
construct: cons, first: car, rest: cdr	(cons 1 (cons 2 nil))
(list 1 2)	
None: nil or'( )	scm> (null? nil) R:#t
malformed list with . (not linked)	
(define x (cons 1 2))	x (1 . 2)
create a list	1. cons 2 list 3'(

1.(cons 1 (cons 2 (cons 3 (cons 4 nil)))) equivalent to :
(list 1 2 3 4)

2. (cons 1 2). equivalent to '(1 2). not (list 1 2)

define lambda expression

(lambda (<formal-parameters>) <body>)

((lambda (x y z) (+ x y (square z))) 1 2 3)

return 12

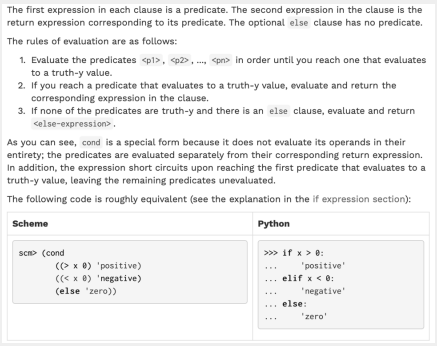
Same as using define, just no name. In fact, the following expressions are equivalent:
(define (plus4 x) (+ x 4))
(define plus4 (lambda (x) (+ x 4)))

!!! REMEMBER: in lambda, paras are in ( ).

cond expression

```
(cond
    (<p1> <e1>)
   (<p1> <e1>)
    ...
   (<pn> <en>)
   [(else <else expression>)])
```

ex: cond



Let

lambda: let

```
(let ( ( <symbol> <expre1>)
    ...
    ( <symbol> <expre2>))
   <body>)
```

Let (cont)

Ex: we can use the approximation sin(x) = x for small x, and sin(x) = 3sin(x/3)-4(sin(x/-3)^3) to approach sin(x) for any x

```
(define (sin x)
   (if (< x 0.000001)
       x
       (let ((recursive-step (sin(/ x 3))))
         (- ( star 3 recursive-step)
             ( star 4 (expt recursive-step
3))))))
```

Lists of Values1

list can contain any data type	Quoting
'(1 2 3)	returns the list (1 2 3)
'(aa bb cc)	returns the list (aa bb cc)
'(1 bb "hello")	returns different data types

1.EVERYTHING in Scheme is a list, even the code.
2. Quoting: to interpret a group of values as a list(instead of as a procedure call), put a single quote in front of them.

special values

booleans	#t #f False false
symbol '	' : don't eval
?	(even? (quotient 45 2)) R: #t
= only work for numbers	(= #t #t) : error
(and)	#t : if you have a partial and, and what value could you and it to, did not change the original value it is true
(or)	#f

1. ' vs " "
(define c 'a)
if call c, return a. if no (define a 1), (eval c) will return error
(define c "a")
if call c, return "a"

2. = only work for numbers
(= #t #t) : error.
(= '(1) '(1)) error

3. but (equal? (= '(1) '(1)) return: true

IF Statement

if takes in two required arguments and an optional third argument:

(if <predicate> <if-true> [if-false])

!! if can do a recursive even at the base case.

if: python vs scheme
1. scheme eval to a value, python directs the flow
2. scheme: if expression just a single expression for each of #t and #f. python: can add more lines
3. scheme: no elif.

scheme vs python

List Operation

Define x to be a list of values(bad)	get the first value of x:
(define x '(1 2 3))	(car)

construct a list from individual values

ex: concat 2 lists

```
(define (concat a b)
 (if (null? a) b
     (cons (car a)
           (concat (cdr a) b)))
)
def concat(a, b):
  if a = Link.empty:
    return b
  else:
    return Link(a.first, concat(a
.rest, b))
```

replicate

```
(define (replicate x n)
  (if (= n 0) nil
      (cons (
  )
)

def replicate(x, n):
  if n == 0:
    return Link.empty
  else:
    return Link(x, replicate(x, n -
1)
```

replicate(5,3)
[5,5,5]
(replicate 5 3)
(5 5 5)

other

and	false finder
or	true finder