Global Definitions

`(define` x E`)`	binds x to E
`(define (`x y_1 `...` y_n`)` E`)`	binds x to proc. E with params. y_1, ..., y

Examples:

```
(define my-number 5)
my-number ⇒ 5
(define x 4)
(define x 7) ;Value of x is 7
(define (square x) (* x x)) ;Function square
(square 5) ⇒ 25
```

What if in `(define` x E`)`, x is a free variable in E? Then get *recursive definition*:

```
(define (fact n) (if zero? n) 1 (* n (fact
(- n 1))))
(fact 3) = (* 3 (fact (- 3 1)))
  = (* 3 (* 2 (fact (- 2 1))))
  = (* 3 (* 2 (* 1 (fact (- 1 1)))))
```


Local Bindings

- Local variables exist in Scheme.
- Lexical (or static) scope, i.e., local variables can be renamed without changing the result of the expression.

Syntax:`(let ((`x_1 E_1`) ... (`x_n E_n`))` E`)`

- Evaluate E_1, E_2, ..., E_k
- Evaluate E with x_i representing the value of E_i
- Return result.

Definitions: `(let ((`x_1 E_1`))` E`)`

- x is called *binding occurrence* (or just a *binding*).
- All occurrences of x in E are said to be within the *scope* of this binding.
- The occurrences of x in E are said to be *bound*.
- Occurrences of x in E_1 are not in the scope of this binding.

Bindings, Cont'd

Why use `let`s?

- Allow subexpressions to be named:

```
(let ((square-3 (square 3))
 (square-4 (square 4)))
 (+ square-3 square-4))
```

- Used to avoid duplicate evaluation

```
(let ((square-3 (square 3)))
   (+ square-3 square-3))

(let ((x 3)) (* x x))
```

- `let` is like passing parameters. Names of formal parameters can be changed and the result remains the same.
- so, "variables can be renamed"

```
(let ((z 3)) (* z z))
(let ((foo 3)) (* foo foo))
```


Bindings (Cont'd)

What is the scope of `x` in

```
(let ((x 2))
   (let ((y 5)) (+ x (* 2 y))))
```

What happens if we have nested bindings of the same variable:

```
(let ((x 2))
   (let ((x (+1 x))) (* x x)))
```

Rename to avoid confusion!

```
(let ((x 2))
   (let ((y (+1 x))) (* y y)))
```

Local bindings "shadow" global, which are then restored

```
(define x 4)
(let ((x 3) (y (+ x 2))) (+ x y)) ⇒
x ⇒ 4
```

Combining Bindings

- `let` evaluates all expressions before binding them
- `let*` is sequential `let`:

(let* ((x_1 E_1) ... (x_n E_n)) E)

- Evaluate E_1 and bind to x_1; evaluate E_2 and bind to x_2...
- Evaluate E with x_i representing the value of E_i
- Return result

```
(define x 0)
(let ((x 2) (y x)) y) ⇒
(let* ((x 2) (y x)) y) ⇒
```


Bindings, Cont'd

```
(define x 4)                  (let*
(define f (lambda () x))        (( x 4)
(define x 5)                    (f (lambda () x))
(f) ⇒ 5                         (x 5))
                              (f)) ⇒ 4
```

- local definitions with `let` are non-recursive
- for local recursive definition, `letrec` form
- allows mutually recursive local definitions:

```
(letrec (
  (odd? (lambda (m) (if (zero? m)
    #f (even? (- m 1)))))
  (even? (lambda (n) (if (zero? n)
    #t (odd? (- n 1)))))))
  (even? 3))
```


Higher-Order Programming

A function is called *higher-order* if its arguments or its result are *functions*. Otherwise, it is *first-order*.

- Higher-order functions permit greater *reuse* of programs
- Instead of writing many specialized functions, we can write a single, generalized function.
- Can do this already to some extent, but higher-order functions take the idea much further.

Example

Here are two specialized but similar *first-order* functions:

```
(define (remove-even L)
  (cond ((null? L) '())
    ((even? (car L)) (remove-even (cdr L)))
    (else (cons (car L) (remove-even (cdr L))))))
```

i.e., remove even elements from list L.

```
(define (remove-odd L)
  (cond ((null? L) '())
    ((odd? (car L)) (remove-odd (cdr L)))
    (else (cons (car L) (remove-odd (cdr L))))))
```

i.e., remove odd elements from list L.

Example (Cont'd)

Here is a more general, *higher-order* function, but with very similar structure:

```
(define (remove-if P L)
  (cond ((null? L) '())
    ((P (car L)) (remove-if P (cdr L)))
    (else (cons (car L) (remove-if P (cdr L))))))
```

i.e., remove from L any element satisfying predicate P.

```
(remove-if even? '(1 2 3)) => (1 3)
(remove-if odd? '(1 2 3)) => (2)
(remove-if number? '(1 a 2 b 3)) => (a b)
```


Another Example: Mapping functions

Here are two specialized but similar *first-order* functions:

```
(define (square-all L)
  (cond ((null? L) '())
        (else (cons (square (car L))
                    (square-all (cdr L))))))
```

i.e., square each element of list L.

```
(define (add1-all L)
  (cond ((null? L) '())
        (else (cons (+1 (car L))
                    (add1-all (cdr L))))))
```

i.e., add 1 to each element of list L.

Function `map`

`map` is a standard, higher-order function that is more general!

```
(define (map F L)
  (cond ((null? L) '())
        (else (cons (F (car L))
                    (map F (cdr L))))))
```

i.e., apply function `F` to every element of list L.

```
(map square '(1 2 3)) => (1 4 9)
(map +1 '(1 2 3)) => (2 3 4)
(map zero? '(0 1 2)) => (#t #f #f)
(map car '((a b) (c d) (e f))) => (a c e)
```


Anonymous (unnamed) functions

When using functions as arguments, it is often inconvenient to have to define them and give them a name:

```
- (define (add2 N) (+ 2 N))
  (map add2 '(1 2 3)) => (3 4 5)
- (define (add3 N) (+ 3 N))
  (map add2 '(1 2 3)) => (4 5 6)
-(define (sq-add6 N) (square (+ 6 N)))
  (map sq-add6 '(1 2 3)) => (49 64 81)
```

Solution: Lambda Expressions

`(lambda (x) (+ 2 X))` is an *anonymous* function that adds 2 to its argument `X`:

```
(map (lambda (X) (+ 2 X)) '(1 2 3 4)) => (3 4 5 6)
(map (lambda (X) (+ 3 X)) '(1 2 3 4)) => (4 5 6 7)
(map (lambda (X) (square (+ 2 X))) '(1 2 3 4))
          => (9 16 25 36)
```

`lambda` expressions can be used anywhere that named functions can be used. For example:

```
((lambda (X) (+ 2 X)) 7) => 9
((lambda (X) (square (+ 2 X))) 3) => 25
(remove-if (lambda (X) (> X 3)) '(1 2 3 4 5 6))
         => (1 2 3)
```


Functions as Values

In Scheme, a function can return another function as its value. For example:

`(define (F X) (if (< 0 X) car cdr))`

Note: `F` always returns `car` or `cdr`, i.e., a function.

```
(F -1) => car
(F 1 ) => cdr
((F -1) '(a b c)) => a
((F 1) '(a b c)) => (b c)
```

Functions can create new functions and return them as lambda expressions. For example:

`(define (compose F G) (lambda (X) (F (G X))))`

i.e. `((compose F G) X) = (F (G X))`.

```
(compose square car) '(3 4 5)) => 9
(map (compose square add2) '(1 2 3 4))
      => (9 16 25 36)
```


Function `reduce`

How do we write a function to add all elements of the list?

```
(define (sum-all l)
  (cond ((null?  l) 0)
    (else (+ (car l) (sum-all (cdr l))))))
```

And to multiply all elements of the list?

```
(define (product-all l)
  (cond ((null?  l) 1)
    (else (* (car l) (product-all (cdr l))))))
```

We want to generalize such functions (REUSE!).

```
(define (reduce f l e)
  (cond ((null?  l) e)
    (else (f (car l) (reduce f (cdr l) e)))))
```

Now,

```
(define (add x y) (+ x y))
(define (multiply x y) (* x y))
(reduce add '(1 2 3 4 5) 0) ⇒ 15
(reduce multiply '(1 2 3 4 5) 1) ⇒ 120
```


How does `reduce` work?

```
(reduce add '(1 2 3) 0)
 = (add 1 (reduce add '(2 3) 0))
 = (add 1 (add 2 (reduce add '(3) 0)))
 = (add 1 (add 2 (add 3 (reduce add '() 0))))
 = (add 1 (add 2 (add 3 0)))
 = (add 1 (add 2 3))
 = (add 1 5)
 = 6
```

In general case, `(reduce f` $(l_1, l_2, ...l_n)$ `v) =`
`(f` l_1 `(f` l_2 `( ...  (f` l_n `v) ...)))`

Creative use of `reduce`

Can define a lot of functions using `reduce`, like `append`, `length`, etc.

How to do that:

1. write the original function, e.g., `length`.
2. `e` is whatever the function returns on base case (i.e., on the empty list).
3. The problem is to write `f`
   (a) Use definition of `reduce` and example to figure out what `f` is supposed to produce
   (b) Write a (named or unnamed) function `f`.

Writing `length` in terms of `reduce`

```
(define (length l)
  (cond ((null?  l) 0)
    (else (+1 (length (cdr l))))))
```

So, `e` is 0.

Now, try `(length '(a b c))` using `reduce`:
`(f a (f b (f c 0)))`:
We want `(f c 0)` to return `1`,
`(f c (f c 0))` to return `2`, and
`(f a (f b (f c 0)))` to return `3`.

So, function `f` should take 2 parameters and return second + 1:
`(lambda (x y) (+1 y))`

Therefore, `length` in terms of `reduce` is:

```
(length L) =
  (reduce (lambda (x y) (+1 y)) L 0)
```


`reduce_if even` in terms of `reduce`

Function `(reduce-if even L)` should return `'()` when `L` is an empty list. So, `e` is `'()`.

Now, try `reduce-if even '(2 5 3 4)` using `reduce`:
`(f 2 (f 5 (f 3 (f 4 '() ))))`:
We want `(f 4 '())` to return '()
(since 4 is even)
`(f 3 (f 4 '()))` to return `(3)`,
`(f 5 (f 3 (f 4 '())))` to return `(5 3)`, and
`(f 2 (f 5 (f 3 (f 4 '()))))` to return `(5 3)`.

So, what is `f`? `f` takes an element and and a list, checks if the element is even, and if not, cons it to the list:

```
(define (temp a y)
  (if (even a) y (cons (a y))))
```

`reduce-if even` in terms of `reduce`:
`(reduce-if even L) = (reduce temp L '())`

Unrestricted Lambda Functions

Goal: want to define functions to take arbitrary number of arguments, e.g.
`(add 1 3 5 7 9)` ⇒ 25
`(add 1 3 5 7 9 11)` ⇒ 36
`(add 1 3 5 7 9 11 13)` ⇒ 39

We do this using *unrestricted* Lambda functions.

Usual form of lambda functions:
`(lambda (`X_1 X_2 `...` X_n`) Body)`

Unrestricted form:
`(lambda X Body)`

Invocation:
`((lambda X Body)` $\texttt{operand}_1$ $\texttt{operand}_2$`...)`

- If the operands $\texttt{operand}_1$ $\texttt{operand}_2$... have the values $\texttt{arg}_1$ $\texttt{arg}_2$ ..., then the variable `X` is a *list* of arguments ($\texttt{arg}_1$ $\texttt{arg}_2$...).
- The body is evaluated with this binding in effect.

Examples of Unrestricted Lambda Functions

• Function `add`

```
(define add
  (letrec (list-add
    (lambda (ls)
      (if (null? ls) 0)
         (+ (car ls) (list-add (cdr ls))))))
  (lambda args (list-add args)))
```

`(add 1 2 3 4 5)` ⇒ 15
Remember that `args` is a list!
• Function `list`

```
(define list (lambda args args))
```

• Another version of `add`

```
(define add
  (lambda args
    (if (null? args) 0
      (+ (car args) (apply add (cdr args))))))
```


Unrestricted Lambda Functions (Cont'd)

In Scheme, a number of functions are defined to take an arbitrary number of arguments: `+`, `*`, `max`, `min`:

`(+ 1 3 5 7 9)` ⇒ 25
`(+ 5)` ⇒ 5
`(+)` ⇒ 0
`(* 2 4 6)` ⇒ 48
`(* 5)` ⇒ 5
`(*)` ⇒ 1

Summary

• *Higher-order* functions can take other functions as arguments and return a function as a value.
• Higher-order functions allow *code reuse*, since a single, general function can replace many specialized functions. (e.g., `remove-if`, `map`)
• Anonymous (or unnamed) functions can be defined with *lambda expressions*.
• Lambda expressions can be used to create new functions during execution.
• General form of lambda expressions:
`(lambda (`X_1 X_2 `...` X_n`) Body)`
• Can define lambda expressions with any number of arguments, with general form:
`(lambda X Body)`

Software Verification

• One of the major problems of software engineering is verifying that a program is correct.
• One approach is to test the program on many different inputs.
• However, subtle bugs may remain undiscovered, only to appear at random, inconvenient, or dangerous moments.
• Another approach is to *prove* that a program is correct for *all* inputs. This is especially useful for safety-critical software (e.g., Air-traffic control systems).

Proving Properties of Programs. Example 1

```
(define (append X Y)
  (if (null? X) Y
      (cons (car X) (append (cdr X) L))))
(define (length X)
  (if (null? X) 0
       (+1 (length (cdr X)))))
```

Theorem:

```
  (length (append X Y)) =
    (length X) + (length Y)
for all lists X, Y
```


Proof outline:

• Use mathematical induction on the length of X.
• First prove that the theorem is true for lists of length 0
Then prove that if the theorem is true for lists of length N, then it is also true for lists of length $N + 1$.
This implies that the theorem is true for lists of any length (i.e., for any list).

Structural Induction

• Actually we will use a variation that emphasizes the *structure* of lists, not their length.
• First prove that the theorem is true for `X = '()`
Then prove that if the theorem is true for `X=L`, then it is also true for `X = (cons E L)`
(note that if L has length N,
then `cons E L` has length $N + 1$)

Preliminaries

• Before using induction (or any other technique) to prove a complex property of a program, write down the basic properties that can be trivially verified by inspecting the program code.
• The inductive proof should only use these properties of the code.
• i.e., the code itself plays no other role in a proof of correctness, and can be henceforth ignored.
• Note: If the basic properties are wrong, then the entire proof is wrong.
So be sure to get them right!!

Basic Properties

```
(define (append X Y)
  (if (null? X) Y
    (cons (car X) (append (cdr X) Y))))

Using X = '(),    (append '()  Y) = Y       (1)
Using X = (cons E L),
   (append (cons E L) Y) =
        (cons E (append L Y))               (2)

(define (length X)
  (if (null? X) 0
      (+1 (length (cdr X)))))
Using X = '(),    (length '()) = 0          (3)
Using X = (cons E L)
   (length (cons E L)) = 1+(length L)       (4)
```

We will use these basic properties to prove (by induction) a more complex property:

```
(length (append X Y))
  = (length X) + (length Y)
```


Proof

• Basis: when `X = '()`

```
  (length (append X Y))
= (length (append '() Y))
= (length Y)                  ;by (1)
= 0 + (length Y)
= (length '()) + (length Y)   ;by (3)
= (length X) + (length Y)
```

So, the theorem is true when `X='()`.

• Inductive Step: Suppose the theorem holds for `X=L`, i.e., suppose that
`(length (append L Y) = (length L) + (length Y)`
This is *inductive hypothesis* (IH).

Proof (Cont'd)

• Now, prove that the theorem holds for `X = (cons E L)`:

```
If X = (cons E L) then
  (length (append X Y))
= (length (append (cons E L) Y))
= (length (cons E (append L Y)))   ; by (2)
= 1 + (length (append L Y))        ; by (4)
= 1 + (length L) + (length Y)      ; by (IH)
= (length (cons E L)) + (length Y) ; by (4)
= (length X) + (length Y), QED.
```

Summary of the proof of theorem:

For any list, `Y`,
Basis: The thm. holds for `X = '()`.
Inductive Step: If the thm holds for `X = L`, then it holds for `X = (cons E L)`.
Thus, by the principle of structural induction, the theorem holds for any lists `X`, `Y`.

Proving Properties of Programs. Example 2

```
(define (member? A X)
  (cond ((null? X) #f)
        ((equal? A (car X)) #t)
        (else (member? A (cdr X)))))
```

Prove the following:
If `(member? A X)`
then `(member? A (append X Y))`
for any `A`, and any lists `X`, `Y`.

Again, will use structural induction on `X` but the prove will be more complex because the program can terminate its recursion in two ways.

Basic properties of `member?`

```
(define (member? A X)
  (cond ((null? X) #f)
        ((equal? A (car X)) #t)
        (else (member? A (cdr X)))))
So, (member? A '()) = #f              (5)
  (member? A (cons A L)) = #t
So, if A = B, then (member? A (cons B L)) = #t
But if A <> B then
  (member? A (cons B L)) = (member? A L)
In general,
  (member? A (cons B L))
     = (A = B) or (member? A L)       (6)
```

We still have properties of `append`:

```
  (append '() Y) = Y                  (1)
  (append (cons E L) Y) =
     (cons E (append L Y))            (2)
```


Proof

Theorem:
If `(member? A X)`
then `(member? A (append X Y))`

Proof (by structural induction on `X`):

- Base case:

```
If X = '() then
  (member? A X)
= (member? A '())
= #f                            ; by (5)
```

So, premise of the thm is false and thus the thm is trivially true (for `X = '()`).

- Inductive step:

Suppose the thm is true for `X = L`, i.e.
If `(member? A L)`
then `(member? A (append L Y))` (IH)

Now, need to prove the theorem is true for `X = (cons E L)`.

Proof (Cont'd)

- Proof of Inductive Step:

```
If (member? A X) then
  (member? A (cons E L))
= (A = E) or (member? A L)           ; by (6)
= (A = E) or (member? A (append L Y)) ; by (IH)
= (member? A (cons E (append L Y)))   ; by (6)
= (member? A (append (cons E L) Y))   ; by (2)
= (member? A (append X Y))
So, for X = (cons E L),
    if (member? A X),
    then (member? A (append X Y)), QED.
```

Summary of the proof of theorem:

For any A and any list Y,
Basis: The thm holds for `X = '()`
Inductive Step: If the thm holds for `X = L`, then it holds for `X = (cons E L)`.
So, by the principle of structural induction, the thm holds for any A and any lists X, Y.

food for slide eater