

## SCHEME

- Read-Eval-Print Loop
- Simple Data Types: booleans, numbers, characters, symbols
- Compound Data Types: strings, vectors, dotted pairs, lists
- **Mantra 1: Every expression has a value**

To find the value of an expression

1. Find values for all subexpressions in any order
2. Apply the value of the first to the values of the rest

## PROCEDURE APPLICATION

- Written in *prefix form*:  
(procedure-name arg1 arg2 ...)
- Prefix notation means that there are no rules for operator precedence or operator associativity
- In a nested expression, innermost expressions are evaluated first. This means the programmer determines the order of evaluation.

$$(+ (* 4 3) (* 5 6)) \Rightarrow 42$$

## LAMBDA

(**lambda** *<parameter list>* *<body>*)

- Lambda creates procedures
- Lambda is a *special form*
- There are three parts to lambda:
  1. lambda
  2. parameter list
  3. body
- The body of the lambda will not be evaluated until the procedure is applied

(**lambda** (*x*) (\* *x x*)) => #<procedure>

((**lambda** (*x*) (\* *x x*)) 4) => 16

(**lambda** (*x y*) (+ *x y*)) => #<procedure>

((**lambda** (*x y*) (+ *x y*)) 2 3) => 5

## DEFINE

```
(define pi 3.1416)
(define square (lambda (x) (* x x)))
pi => 3.1416
square => #<procedure:square>
(square 16) => 256
```

- Identifiers are delimited by whitespace
- No length limit.
- Case insensitive. Can't start with numeric, +, -, or .

A shorthand for defining procedures:

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

(define (hello-world)
  (display "Hello, World!")
  (newline))
```

## BOOLEANS

- Boolean value for true is #t and false is #f
- Scheme treats #f as false and everything else as true.
- **Mantra 2:**  
**Everything that isn't #f is true.**

(**and** <exp1> <exp2> ...)

And evaluates the expressions one at a time from left to right. As soon as one of the expressions evaluates to #f (false), the value #f is returned and none of the remaining expressions are evaluated.

(**or** <exp1> <exp2> ...)

Or evaluates the expressions one at a time from left to right. As soon as one of the expressions evaluates to true, the value is returned and evaluation of remaining expressions is terminated.

## LISTS

- Lists are sequences of objects surrounded by parentheses.

- Examples:

(1 2 3 4)

("hci" 121 "scheme")

(a b c (d e f))

- A procedure looks just like a list.
- To treat a list as data rather than as a procedure application use *quote*

- Examples:

(**quote** (+ 3 4))=> (+ 3 4)

But would write as:

'(+ 3 4)

## LIST MANIPULATION PROCEDURES

- *car* returns the first element of a list
- *cdr* returns the remainder of a list

$(car \text{ 'this is a list}) \Rightarrow this$

$(cdr \text{ 'this is a list}) \Rightarrow (is \ a \ list)$

- *cons* constructs lists by adding a new element to the beginning of a list

$(cons \ \text{first} \ \text{'among many}) \Rightarrow (first \ among \ many)$

- the procedure *list* takes any number of arguments and builds a list

$(list \ \text{this} \ \text{'is} \ \text{'a} \ \text{'list}) \Rightarrow (this \ is \ a \ list)$

$(list) \Rightarrow ()$

$(list \ \text{a} \ \text{'(b c)}) \Rightarrow (a \ (b \ c))$

## PROPER and IMPROPER LISTS and DOTTED PAIRS

- A *list* is a sequence of *pairs*
- Each *pairs's* `cdr` is the next pair
- The `cdr` of the last pair in a *proper list* is the empty list.
- If the `cdr` of the last pair is not the empty list, the list is an *improper list*
- Lists are made of *dotted-pairs*

$(\text{cons } 'a \ 'b) \Rightarrow (a . b)$

$(\text{cdr } '(a . b)) \Rightarrow b$

$(\text{cons } 'a \ '()) \Rightarrow (a)$

- Null list  $()$

$(\text{null? } '()) \Rightarrow \#t$

$(\text{null? } '(scheme)) \Rightarrow \#f$

## SETTING VARIABLES

- Use *define* to create a variable and assign a value
- Scheme has the convention of ending procedure names with *!* if they cause side effects.
- Another naming convention is to end predicates (procedures that return boolean values) with *?*
- Use *set!* to change the value of a variable
- Use *let* to temporarily create a variable and assign a value.
- Let expressions include a list of *variable-value pairs* and a sequence of expressions called the *body*

```
(let ((x 1) (y 2))  
  (+ x y)) => 3
```

## LET EXPRESSIONS

- Procedures are no different than any other list element, so we can do this:

**(let ((f +))**

**(f 1 2)) => 3**

**(let ((+ \*))**

**(+ 2 3)) => 6**

- Values bound by *let* are only *visible* within the body
- Lets can be nested

**(let ((x 1))**

**(let ((x (+ x 1)))**

**(+ x x))) => 4**

- The scope of each variable can be determined by it's placement. This is called *lexical scoping*

## QUIZ

- What is the value of:

```
(let ((x 9))  
  (* x  
    (let ((x (/ x 3)))  
      (+ x x)))) => ?
```

- What is the value of:

```
(let ((foo (lambda (x) (+ x 1))))  
  
  (let ((y 3))  
    (foo y)))=> ?
```

## LAMBDA EXAMPLES

```
(let ((double (lambda (x) (+ x x))))  
  (list (double (* 3 4))  
        (double (/ 99 11))  
        (double (- 2 7))))  
=> (24 18 -10)
```

```
(let ((double-cons  
      (lambda (x) (cons x x))))  
  (double-cons 'a))  
=> (a . a)
```

We can pass a procedure as a parameter!

```
(let ((double-any  
      (lambda (f x) (f x x))))  
  (list (double-any + 13)  
        (double-any cons 'a))) => (26 (a . a))
```

## LAMBDA PARAMETERS

- There are three possible forms for the formal parameter specification in a lambda expression:
  1. The formal parameters may be null (lambda () ...)
  2. The formal parameters may be a proper list of variables (lambda (x) ...) (lambda (a b c) ...)
  3. The formal parameters may be an improper list of variables (lambda (x y . z) ...)
- Examples:

$((\mathbf{lambda} (x\ y\ .\ z) (list\ x\ y\ z))\ 1\ 2\ 3\ 4\ 5\ 6\ 7)$   
 $\Rightarrow (1\ 2\ (3\ 4\ 5\ 6\ 7))$

## CONTROL

- Procedure Application

- Sequencing

(**begin** *exp1 exp2 ...*)

- Conditionals

(**if** *test consequent alternative*)

(*not exp*)

(**and** *exp*)

(**or** *exp*)

(**cond** *clause1 clause2 ...*)

(**case** *exp clause1 clause2 ...*)

## EXAMPLES

`car` and `cdr` are the basic primitives for dealing with lists

Suppose we want the second element of a list

$(\text{car } (\text{cdr } '(a\ b\ c))) \Rightarrow b$

These can be abbreviated. (`car` (`cdr` is `cadr`

$(\text{cadr } '(a\ b\ c)) \Rightarrow b$

The third element is

$(\text{car } (\text{cdr } (\text{cdr } '(a\ b\ c)))) \Rightarrow c$

or

$(\text{caddr } '(a\ b\ c)) \Rightarrow c$

$(\text{caddr } '(a\ b\ c\ d\ e\ f)) \Rightarrow (c\ d\ e\ f)$

$(\text{cadddr } '(a\ b\ c\ d\ e\ f)) \Rightarrow d$

$(\text{car } (\text{cdr } (\text{cdr } (\text{cdr } '(a\ b\ c\ d\ e\ f)))))) \Rightarrow d$

$(\text{list } (+\ 3\ 4)\ 'scheme\ +\ \#\text{f } (\text{list } 'human\ 'computer\ 'interaction)\ 'hci) \Rightarrow (7\ scheme\ \#\langle\text{primitive:}+\rangle\ \#\text{f } (human\ computer\ interaction))$

The function *append* takes any number of lists as arguments and appends them into a list.

$(\text{append } (\text{list } '(a\ b)\ 'c)\ '(d\ e\ f)) \Rightarrow ((a\ b)\ c\ d\ e\ f)$

$(\text{append } (\text{list } (\text{list } 'a\ 'b)\ 'c)\ (\text{list } 'd\ 'e\ 'f)) \Rightarrow ((a\ b)\ c\ d\ e\ f)$

*(map proc list1 list2 ...)*

The function *map* applies *proc* to each element of the list and returns a list of the results. If more than one list is given, then they must all be the same length. The dynamic order in which *proc* is applied to the elements of the list is unspecified. ;

*(map car '(john lennon) (paul mccartney) (george harrison) (ringo star))*  
*=> (john paul george ringo)*

The function *for-each* calls *proc* for its side effects rather than for its values. Unlike *map*, *for-each* is guaranteed to call *proc* on the elements of the lists in order from the first element to the last. The value returned by *for-each* is unspecified.

*(for-each proc list1 list2 ...)*

## RECURSION

```
(define (my-reverse ls)  
  (if (null? ls)  
    '()  
    (append (my-reverse (cdr ls)) (list (car ls))))))
```

*(my-reverse '(a (b c) d)) => (d (b c) a)*

```
(define (remove item ls)  
  (cond  
    ((null? ls) '())  
    ((equal? (car ls) item) (remove item (cdr ls)))  
    (else (cons (car ls) (remove item (cdr ls))))))
```

*(remove 'a '(a b c a)) => (b c)*

But notice *my-reverse* and *remove* only work at top level

*(remove 'a (list '(a b) 'c 'd)) => ((a b) c d)*

```
(require-library "trace.ss" "mzlib")
(trace my-reverse)
(my-reverse '(a b c))
|(my-reverse (a b c))
| (my-reverse (b c))
| |(my-reverse (c))
| | (my-reverse ())
| | ()
| |(c)
| (c b)
|(c b a)
(c b a)
```

```
(trace remove)
(remove)
(remove 'a '(a b c a))
|(remove a (a b c a))
| (remove a (b c a))
| |(remove a (c a))
| | (remove a (a))
| | |(remove a ())
| | |()
| | ()
| |(c)
| (b c)
|(b c)
(b c)
```

A recursive procedure typically has the following:

- a *base case* that will terminate the recursion
- a *recursive case* that makes the recursive call on a simpler version of the expression

There is a built in function *length* but let's implement our version.

```
(length '(a b c))
```

```
3
```

```
(define (my-length lis)
  (cond ((null? lis)
         0)
        (else
         (+ 1 (my-length (cdr lis))))))
```

```
> (trace my-length)
> (my-length)
> (my-length '(a b c))
|(my-length (a b c))
| (my-length (b c))
| |(my-length (c))
| | (my-length ())
| | 0
| | 1
| | 2
| | 3
3
>
```

```
> (my-length '())
|(my-length ())
|0
0
> (my-length '(a (a b) c))
|(my-length (a (a b) c))
| (my-length ((a b) c))
| |(my-length (c))
| | (my-length ())
| | 0
| | 1
| 2
| 3
3
>
```

```
(define (remove-1st item ls)  
  (cond  
    ((null? ls) '())  
    ((equal? (car ls) item) (cdr ls))  
    ((else (cons (car ls) (remove-1st item (cdr ls))))))))
```

```
> (remove-1st 'a '(a b c d a))
```

```
(b c d a)
```

```
> (trace remove-1st)
```

```
(remove-1st)
```

```
> (remove-1st 'a '(a b c d a))
```

```
|(remove-1st a (a b c d a))
```

```
|(b c d a)
```

```
(b c d a)
```

```
(define (remove item ls)  
  (cond  
    ((null? ls) '())  
    ((equal? (car ls) item) (remove item (cdr ls)))  
    (else (cons (car ls) (remove item (cdr ls))))))  
  
> (remove 'a '(a b c d a))  
(b c d)
```

```
> (remove 'a '(a b c d a))
|(remove a (a b c d a))
| (remove a (b c d a))
| |(remove a (c d a))
| | (remove a (d a))
| | |(remove a (a))
| | | (remove a ())
| | | ()
| | |()
| | (d)
| |(c d)
| (b c d)
|(b c d)
(b c d)
>
```

```
(define (remove-all item ls)  
  (cond  
    ((null? ls) '())  
    ((equal? (car ls) item) (remove-all item (cdr ls)))  
    ((pair? (car ls)) (cons (remove-all item (car ls))  
                           (remove-all item (cdr ls))))  
    (else (cons (car ls) (remove-all item (cdr ls))))))
```

```
> (remove-all 'a '(a (a b) c d))  
((b) c d)  
> (trace remove-all)
```

```

(remove-all)
> (remove-all 'a '(a (a b) c d))
|(remove-all a (a (a b) c d))
| (remove-all a ((a b) c d))
| |(remove-all a (a b))
| | (remove-all a (b))
| | |(remove-all a ())
| | |()
| | |(b)
| | |(b)
| |(remove-all a (c d))
| | (remove-all a (d))
| | |(remove-all a ())
| | |()
| | |(d)
| |(c d)
| ((b) c d)
|((b) c d)
((b) c d)
>

```

```
(define (flatten ls)
  (cond
    ((null? ls) '())
    ((pair? (car ls))
     (append (flatten (car ls)) (flatten (cdr ls))))
    (else (cons (car ls) (flatten (cdr ls))))))

> (flatten '(a (a (b c)) d))
(a a b c d)
```

```

> (flatten '(a (a (b c)) d))
|(flatten (a (a (b c)) d))
| (flatten ((a (b c)) d))
| |(flatten (a (b c)))
| | (flatten ((b c)))
| | |(flatten (b c))
| | | (flatten (c))
| | | |(flatten ())
| | | |()
| | | (c)
| | |(b c)
| | |(flatten ())
| | |()
| | (b c)
| |(a b c)
| |(flatten (d))
| | (flatten ())
| | ()
| |(d)
| (a b c d)
|(a a b c d)
(a a b c d)
>

```