READ-EVAL-PRINT Loop

READ: Read input from user:
a procedure application

EVAL: Evaluate input:
`(f arg`$_1$ `arg`$_2$ `...arg`$_n$`)`
1. evaluate `f` to obtain a procedure
2. evaluate each `arg`$_i$ to obtain a value
3. apply procedure to argument values

PRINT: Print resulting value:
the result of the procedure application

READ-EVAL-PRINT Loop Example

```
1 ]=> (cons 'a (cons 'b '(c d)))
;Value 1: (a b c d)
```

1. Read the procedure application
   `(cons 'a (cons 'b '(c d)))`
2. Evaluate `cons` to obtain a procedure
3. Evaluate `'a` to obtain `a` itself
4. Evaluate `(cons 'b '(c d))`:
   (a) Evaluate `cons` to obtain a procedure
   (b) Evaluate `'b` to obtain `b` itself
   (c) Evaluate `'(c d)` to obtain `(c d)` itself
   (d) Apply the `cons` procedure to `b` and `(c d)` to obtain `(b c d)`
5. Apply the `cons` procedure to `a` and `(b c d)` to obtain `(a b c d)`
6. Print the result of the application:
   `(a b c d)`

Quotes Inhibit Evaluation

```
;;Same as before:
1 ]=> (cons 'a (cons 'b '(c d)))
;Value 2: (a b c d)

;;Now quote the second argument:
1 ]=> (cons 'a '(cons 'b '(c d)))
;Value 3: (a cons (quote b) (quote (c d)))

;;Instead, un-quote the first argument:
1 ]=> (cons a (cons 'b '(c d)))
;Unbound variable:  a
;To continue, call RESTART...
2 error> ^C^C
1 ]=>
```


Quotes vs. Eval

```
;;Some things evaluate to themselves:
1 ]=> (list 1 42 #t #f ())
;Value 4: (1 2 #t () ())

;;They can also be quoted:
1 ]=> (list '1 '42 '#t '#f '())
;Value 5: (1 2 #t () ())
```

Eval Activates Evaluation

```
1 ]=> '(+ 1 2)
;Value 6: (+ 1 2)

;;Eval can be used to evaluate an expression
1 ]=> (eval '(+ 1 2))
;Value 7: 3
```

READ-EVAL-PRINT Loop

Can also be used to define procedures.

READ: Read input from user:
a symbol definition

EVAL: Evaluate input:
store function definition

PRINT: Print resulting value:
the symbol defined

Example:

```
1 ]=> (define (square x) (* x x))

;Value: square
```


Procedure Definition

Two syntaxes for definition:

1.
```
(define (<fcn-name> <fcn-params>)
<expression>)
(define (square x)
  (* x x))

(define (mean x y)
  (/ (+ x y) 2))
```

2.
```
(define <fcn-name> <fcn-value>)

(define square
  (lambda (n) (* n n)))

(define mean
  (lambda (x y) (/ (+ x y) 2)))
```

Lambda procedure syntax enables the creation of anonymous procedures. More on this later!

Conditional Execution: if

```
(if <condition> <result1> <result2>)
```

1. Evaluate `<condition>`
2. If the result is a "true value" (i.e., anything but () or `#f`), then evaluate and return `<result1>`
3. Otherwise, evaluate and return `<result2>`

```
(define (abs-val x)
  (if (>= x 0) x (- x)))

(define (rest-if-first e lst)
  (if (eq? e (car lst)) (cdr lst) '()))
```


Conditional Execution: cond

```
(cond (<condition1> <result1>)
      (<condition2> <result2>)
      ...
      (<conditionN> <resultN>)
      (else <else-result>) ;optional else
)                          ;clause
```

1. Evaluate conditions in order until obtaining one that returns a true value
2. Evaluate and return the corresponding result
3. If none of the conditions returns a true value, evaluate and return `<else-result>`

Conditional Execution: cond

```
(define (abs-val x)
        (cond ((>= x 0) x)
              (else (- x))
         )
 )

(define (rest-if-first e lst)
        (cond ((null? lst) '())
              ((eq? e (car lst)) (cdr lst))
              (else '())
         )
 )
```


Conditional vs. Boolean Expressions

Write a procedure that takes a parameter `x` and returns `#t` if `x` is an atom, and false otherwise. Using `cond`:

```
(define (atom? x)
  (cond ((symbol? x) '#t)
        ((number? x) '#t)
        ((char? x) '#t)
        ((string? x) '#t)
        ((null? x) '#t)
        (else ())
  )
)
```

Conditional vs. Boolean Expressions

Now write `atom?` without using `cond`:

```
(define (atom? x)
  (if (symbol? x) '#t
    (if (number? x) '#t
      (if (char? x) '#t
        (if (string? x) '#t
          (if (null? x) '#t () )
        )
      )
    )
  )
)
```


Better atom? procedure

Any list is a pair (dotted pair with CAR and CDR), except the empty list (which is both list and atom).

```
(define (atom? x)
  (if (pair? x) () '#t)
)

(define (atom? x)
  (cond ((pair? x) ())
        (else '#t)
  )
)
```

Recursion: Five Steps to a Recursive Function

1. **Strategy:** How to reduce the problem?
2. **Header:**
   - What info needed as input and output?
   - Write the function header.
     Use a noun phrase for the function name.
3. **Spec:** Write a method specification in terms of the parameters and return value.
   Include preconditions.
4. **Base Cases:**
   - When is the answer so simple that we know it without recursing?
   - What is the answer in these base case(s)?
   - Write code for the base case(s).
5. **Recursive Cases:**
   - Describe the answer in the other case(s) in terms of the answer on smaller inputs.
   - Simplify if possible.
   - Write code for the recursive case(s).

Recursive Scheme Procedures: Sum-N

Parameter: integer n ≥ 0.

Result: sum of integers from 0 to n.

```
(define (sum-n n)

  (cond (                              )


        (else                          )

   )
 )
```

Recursive Scheme Procedures: Length

```
(define (length x)








   ))
```

This is called "cdr-recursion."

Note: There is a built-in `length` procedure.

Length (cont.)

```
1 ]=> (trace length)

;No value

1 ]=> (length '(a b c))

[Entering #[compound-procedure 5 length]
    Args: (a b c)]
[Entering #[compound-procedure 5 length]
    Args: (b c)]
[Entering #[compound-procedure 5 length]
    Args: (c)]
[Entering #[compound-procedure 5 length]
    Args: ()]
[0
      <== #[compound-procedure 5 length]
    Args: ()]
[1
      <== #[compound-procedure 5 length]
    Args: (c)]
[2
      <== #[compound-procedure 5 length]
    Args: (b c)]
[3
      <== #[compound-procedure 5 length]
    Args: (a b c)]
;Value: 3
```

Recursive Scheme Procedures: Abs-List

- (abs-list '(1 -2 -3 4 0)) ⇒ (1 2 3 4 0)
- (abs-list '()) ⇒ ()

```
(define (abs-list lst)







)
```


Recursive Scheme Procedures: Append

```
(append '(1 2) '(3 4 5)) ⇒ (1 2 3 4 5)
(append '(1 2) '(3 (4) 5)) ⇒ (1 2 3 (4) 5)
(append '() '(1 4 5)) ⇒ (1 4 5)
(append '(1 4 5) '()) ⇒ (1 4 5)
(append '() '()) ⇒ ()

(define (append x y)





)
```

Note: There is a built-in **append** procedure.