---

language: Paren filename: learnparen.paren contributors: - [“KIM Taegyoon”, “https://github.com/kimtg”]
—

Paren is a dialect of Lisp. It is designed to be an embedded language.

Some examples are from http://learnxinyminutes.com/docs/racket/.

```
;;; Comments
# comments

;; Single line comments start with a semicolon or a sharp sign

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 1. Primitive Datatypes and Operators
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; Numbers
123 ; int
3.14 ; double
6.02e+23 ; double
(int 3.14) ; => 3 : int
(double 123) ; => 123 : double

;; Function application is written (f x y z ...)
;; where f is a function and x, y, z, ... are operands
;; If you want to create a literal list of data, use (quote) to stop it from
;; being evaluated
(quote (+ 1 2)) ; => (+ 1 2)
;; Now, some arithmetic operations
(+ 1 1)  ; => 2
(- 8 1)  ; => 7
(* 10 2) ; => 20
(^ 2 3) ; => 8
(/ 5 2) ; => 2
(% 5 2) ; => 1
(/ 5.0 2) ; => 2.5

;;; Booleans
true ; for true
false ; for false
(! true) ; => false
(&& true false (prn "doesn't get here")) ; => false
(|| false true (prn "doesn't get here")) ; => true

;;; Characters are ints.
(char-at "A" 0) ; => 65
(chr 65) ; => "A"

;;; Strings are fixed-length array of characters.
"Hello, world!"
"Benjamin \"Bugsy\" Siegel"   ; backslash is an escaping character
"Foo\tbar\r\n" ; includes C escapes: \t \r \n
```

```
;; Strings can be added too!
(strcat "Hello " "world!") ; => "Hello world!"

;; A string can be treated like a list of characters
(char-at "Apple" 0) ; => 65

;; Printing is pretty easy
(pr "I'm" "Paren. ") (prn "Nice to meet you!")

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 2. Variables
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; You can create or set a variable using (set)
;; a variable name can use any character except: ();#"
(set some-var 5) ; => 5
some-var ; => 5

;; Accessing a previously unassigned variable is an exception
; x ; => Unknown variable: x : nil

;; Local binding: Use lambda calculus! `a' and `b' are bound to `1' and `2' only within the (fn ...)
((fn (a b) (+ a b)) 1 2) ; => 3

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 3. Collections
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; Lists

;; Lists are vector-like data structures. (Random access is O(1).)
(cons 1 (cons 2 (cons 3 (list)))) ; => (1 2 3)
;; `list' is a convenience variadic constructor for lists
(list 1 2 3) ; => (1 2 3)
;; and a quote can also be used for a literal list value
(quote (+ 1 2)) ; => (+ 1 2)

;; Can still use `cons' to add an item to the beginning of a list
(cons 0 (list 1 2 3)) ; => (0 1 2 3)

;; Lists are a very basic type, so there is a *lot* of functionality for
;; them, a few examples:
(map inc (list 1 2 3))          ; => (2 3 4)
(filter (fn (x) (== 0 (% x 2))) (list 1 2 3 4))    ; => (2 4)
(length (list 1 2 3 4))     ; => 4

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 3. Functions
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Use `fn' to create functions.
;; A function always returns the value of its last expression
(fn () "Hello World") ; => (fn () Hello World) : fn

;; Use parentheses to call all functions, including a lambda expression
```

```
((fn () "Hello World")) ; => "Hello World"

;; Assign a function to a var
(set hello-world (fn () "Hello World"))
(hello-world) ; => "Hello World"

;; You can shorten this using the function definition syntatcic sugae:
(defn hello-world2 () "Hello World")

;; The () in the above is the list of arguments for the function
(set hello
  (fn (name)
    (strcat "Hello " name)))
(hello "Steve") ; => "Hello Steve"

;; ... or equivalently, using a sugared definition:
(defn hello2 (name)
  (strcat "Hello " name))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 4. Equality
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; for numbers use `=='
(== 3 3.0) ; => true
(== 2 1) ; => false

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 5. Control Flow
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; Conditionals

(if true                ; test expression
    "this is true"    ; then expression
    "this is false") ; else expression
; => "this is true"

;;; Loops

;; for loop is for number
;; (for SYMBOL START END STEP EXPR ..)
(for i 0 10 2 (pr i "")) ; => prints 0 2 4 6 8 10
(for i 0.0 10 2.5 (pr i "")) ; => prints 0 2.5 5 7.5 10

;; while loop
((fn (i)
  (while (< i 10)
    (pr i)
    (++ i))) 0) ; => prints 0123456789

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 6. Mutation
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;; Use `set' to assign a new value to a variable or a place
(set n 5) ; => 5
(set n (inc n)) ; => 6
n ; => 6
(set a (list 1 2)) ; => (1 2)
(set (nth 0 a) 3) ; => 3
a ; => (3 2)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 7. Macros
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Macros let you extend the syntax of the language.
;; Paren macros are easy.
;; In fact, (defn) is a macro.
(defmacro setfn (name ...) (set name (fn ...)))
(defmacro defn (name ...) (def name (fn ...)))

;; Let's add an infix notation
(defmacro infix (a op ...) (op a ...))
(infix 1 + 2 (infix 3 * 4)) ; => 15

;; Macros are not hygienic, you can clobber existing variables!
;; They are code transformations.
```