#lang racket ; defines the language we are using  
  
;;; Comments  
  
;; Single line comments start with a semicolon  
  
#| Block comments  
 can span multiple lines and...  
 #|  
 they can be nested!  
 |#  
|#  
  
;; S-expression comments discard the following expression,  
;; useful to comment expressions when debugging  
#; (this expression is discarded)  
  
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;  
;; 1. Primitive Datatypes and Operators  
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;  
  
;;; Numbers  
9999999999999999999999 ; integers  
#b111 ; binary => 7  
#o111 ; octal => 73  
#x111 ; hexadecimal => 273  
3.14 ; reals  
6.02e+23  
1/2 ; rationals  
1+2i ; complex numbers  
  
;; 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 ' to stop it from  
;; being evaluated  
'(+ 1 2) ; => (+ 1 2)  
;; Now, some arithmetic operations  
(+ 1 1) ; => 2  
(- 8 1) ; => 7  
(\* 10 2) ; => 20  
(expt 2 3) ; => 8  
(quotient 5 2) ; => 2  
(remainder 5 2) ; => 1  
(/ 35 5) ; => 7  
(/ 1 3) ; => 1/3  
(exact->inexact 1/3) ; => 0.3333333333333333  
(+ 1+2i 2-3i) ; => 3-1i  
  
;;; Booleans  
#t ; for true  
#f ; for false -- any value other than #f is true  
(not #t) ; => #f  
(and 0 #f (error "doesn't get here")) ; => #f  
(or #f 0 (error "doesn't get here")) ; => 0  
  
;;; Characters  
#\A ; => #\A  
#\λ ; => #\λ  
#\u03BB ; => #\λ  
  
;;; Strings are fixed-length array of characters.  
"Hello, world!"  
"Benjamin \"Bugsy\" Siegel" ; backslash is an escaping character  
"Foo\tbar\41\x21\u0021\a\r\n" ; includes C escapes, Unicode  
"λx:(μα.α→α).xx" ; can include Unicode characters  
  
;; Strings can be added too!  
(string-append "Hello " "world!") ; => "Hello world!"  
  
;; A string can be treated like a list of characters  
(string-ref "Apple" 0) ; => #\A  
  
;; format can be used to format strings:  
(format "~a can be ~a" "strings" "formatted")  
  
;; Printing is pretty easy  
(printf "I'm Racket. Nice to meet you!\n")  
  
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;  
;; 2. Variables  
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;  
;; You can create a variable using define  
;; a variable name can use any character except: ()[]{}",'`;#|\  
(define some-var 5)  
some-var ; => 5  
  
;; You can also use unicode characters  
(define ⊆ subset?)  
(⊆ (set 3 2) (set 1 2 3)) ; => #t  
  
;; Accessing a previously unassigned variable is an exception  
; x ; => x: undefined ...  
  
;; Local binding: `me' is bound to "Bob" only within the (let ...)  
(let ([me "Bob"])  
 "Alice"  
 me) ; => "Bob"  
  
;; let\* is like let, but allows you to use previous bindings in creating later bindings  
(let\* ([x 1]  
 [y (+ x 1)])  
 (\* x y))  
  
;; finally, letrec allows you to define recursive and mutually recursive functions  
(letrec ([is-even? (lambda (n)  
 (or (zero? n)  
 (is-odd? (sub1 n))))]  
 [is-odd? (lambda (n)  
 (and (not (zero? n))  
 (is-even? (sub1 n))))])  
 (is-odd? 11))  
  
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;  
;; 3. Structs and Collections  
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;  
  
;; Structs  
; By default, structs are immutable  
(struct dog (name breed age))  
(define my-pet  
 (dog "lassie" "collie" 5))  
my-pet ; => #<dog>  
; returns whether the variable was constructed with the dog constructor  
(dog? my-pet) ; => #t  
; accesses the name field of the variable constructed with the dog constructor  
(dog-name my-pet) ; => "lassie"  
  
; You can explicitly declare a struct to be mutable with the #:mutable option  
(struct rgba-color (red green blue alpha) #:mutable)  
(define burgundy  
 (rgba-color 144 0 32 1.0))  
(set-rgba-color-green! burgundy 10)  
(rgba-color-green burgundy) ; => 10  
  
;;; Pairs (immutable)  
;; `cons' constructs pairs, `car' and `cdr' extract the first  
;; and second elements  
(cons 1 2) ; => '(1 . 2)  
(car (cons 1 2)) ; => 1  
(cdr (cons 1 2)) ; => 2  
  
;;; Lists  
  
;; Lists are linked-list data structures, made of `cons' pairs and end  
;; with a `null' (or '()) to mark the end of the list  
(cons 1 (cons 2 (cons 3 null))) ; => '(1 2 3)  
;; `list' is a convenience variadic constructor for lists  
(list 1 2 3) ; => '(1 2 3)  
;; a quote can also be used for a literal list value  
'(1 2 3) ; => '(1 2 3)  
;; a quasiquote (represented by the backtick character) with commas   
;; can be used to evaluate functions  
`(1 ,(+ 1 1) 3) ; => '(1 2 3)  
  
;; With lists, car/cdr work slightly differently  
(car '(1 2 3)) ; => 1  
(cdr '(1 2 3)) ; => '(2 3)  
  
;; Racket also has predefined functions on top of car and cdr, to extract parts of a list  
(cadr (list 1 2 3)) ; => 2  
(car (cdr (list 1 2 3))) ; => 2  
  
(cddr (list 1 2 3)) ; => '(3)  
(cdr (cdr (list 1 2 3))) ; => '(3)  
  
(caddr (list 1 2 3)) ; => 3  
(car (cdr (cdr (list 1 2 3)))) ; => 3  
  
;; Can still use `cons' to add an item to the beginning of a list  
(cons 4 '(1 2 3)) ; => '(4 1 2 3)  
  
;; Use `append' to add lists together  
(append '(1 2) '(3 4)) ; => '(1 2 3 4)  
  
;; Lists are a very basic type, so there is a \*lot\* of functionality for  
;; them, a few examples:  
(map add1 '(1 2 3)) ; => '(2 3 4)  
(map + '(1 2 3) '(10 20 30)) ; => '(11 22 33)  
(filter even? '(1 2 3 4)) ; => '(2 4)  
(count even? '(1 2 3 4)) ; => 2  
(take '(1 2 3 4) 2) ; => '(1 2)  
(drop '(1 2 3 4) 2) ; => '(3 4)  
  
;;; Vectors  
  
;; Vectors are fixed-length arrays  
#(1 2 3) ; => '#(1 2 3)  
  
;; Use `vector-append' to add vectors together  
(vector-append #(1 2 3) #(4 5 6)) ; => #(1 2 3 4 5 6)  
  
;;; Sets  
  
;; Create a set from a list  
(list->set '(1 2 3 1 2 3 3 2 1 3 2 1)) ; => (set 1 2 3)  
  
;; Add a member with `set-add'  
;; (Functional: returns the extended set rather than mutate the input)  
(set-add (set 1 2 3) 4) ; => (set 1 2 3 4)  
  
;; Remove one with `set-remove'  
(set-remove (set 1 2 3) 1) ; => (set 2 3)  
  
;; Test for existence with `set-member?'  
(set-member? (set 1 2 3) 1) ; => #t  
(set-member? (set 1 2 3) 4) ; => #f  
  
;;; Hashes  
  
;; Create an immutable hash table (mutable example below)  
(define m (hash 'a 1 'b 2 'c 3))  
  
;; Retrieve a value  
(hash-ref m 'a) ; => 1  
  
;; Retrieving a non-present value is an exception  
; (hash-ref m 'd) => no value found  
  
;; You can provide a default value for missing keys  
(hash-ref m 'd 0) ; => 0  
  
;; Use `hash-set' to extend an immutable hash table  
;; (Returns the extended hash instead of mutating it)  
(define m2 (hash-set m 'd 4))  
m2 ; => '#hash((b . 2) (a . 1) (d . 4) (c . 3))  
  
;; Remember, these hashes are immutable!  
m ; => '#hash((b . 2) (a . 1) (c . 3)) <-- no `d'  
  
;; Use `hash-remove' to remove keys (functional too)  
(hash-remove m 'a) ; => '#hash((b . 2) (c . 3))  
  
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;  
;; 3. Functions  
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;  
  
;; Use `lambda' to create functions.  
;; A function always returns the value of its last expression  
(lambda () "Hello World") ; => #<procedure>  
;; Can also use a unicode `λ'  
(λ () "Hello World") ; => same function  
  
;; Use parens to call all functions, including a lambda expression  
((lambda () "Hello World")) ; => "Hello World"  
((λ () "Hello World")) ; => "Hello World"  
  
;; Assign a function to a var  
(define hello-world (lambda () "Hello World"))  
(hello-world) ; => "Hello World"  
  
;; You can shorten this using the function definition syntactic sugar:  
(define (hello-world2) "Hello World")  
  
;; The () in the above is the list of arguments for the function  
(define hello  
 (lambda (name)  
 (string-append "Hello " name)))  
(hello "Steve") ; => "Hello Steve"  
;; ... or equivalently, using a sugared definition:  
(define (hello2 name)  
 (string-append "Hello " name))  
  
;; You can have multi-variadic functions too, using `case-lambda'  
(define hello3  
 (case-lambda  
 [() "Hello World"]  
 [(name) (string-append "Hello " name)]))  
(hello3 "Jake") ; => "Hello Jake"  
(hello3) ; => "Hello World"  
;; ... or specify optional arguments with a default value expression  
(define (hello4 [name "World"])  
 (string-append "Hello " name))  
  
;; Functions can pack extra arguments up in a list  
(define (count-args . args)  
 (format "You passed ~a args: ~a" (length args) args))  
(count-args 1 2 3) ; => "You passed 3 args: (1 2 3)"  
;; ... or with the unsugared `lambda' form:  
(define count-args2  
 (lambda args  
 (format "You passed ~a args: ~a" (length args) args)))  
  
;; You can mix regular and packed arguments  
(define (hello-count name . args)  
 (format "Hello ~a, you passed ~a extra args" name (length args)))  
(hello-count "Finn" 1 2 3)  
; => "Hello Finn, you passed 3 extra args"  
;; ... unsugared:  
(define hello-count2  
 (lambda (name . args)  
 (format "Hello ~a, you passed ~a extra args" name (length args))))  
  
;; And with keywords  
(define (hello-k #:name [name "World"] #:greeting [g "Hello"] . args)  
 (format "~a ~a, ~a extra args" g name (length args)))  
(hello-k) ; => "Hello World, 0 extra args"  
(hello-k 1 2 3) ; => "Hello World, 3 extra args"  
(hello-k #:greeting "Hi") ; => "Hi World, 0 extra args"  
(hello-k #:name "Finn" #:greeting "Hey") ; => "Hey Finn, 0 extra args"  
(hello-k 1 2 3 #:greeting "Hi" #:name "Finn" 4 5 6)  
 ; => "Hi Finn, 6 extra args"  
  
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;  
;; 4. Equality  
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;  
  
;; for numbers use `='  
(= 3 3.0) ; => #t  
(= 2 1) ; => #f  
  
;; `eq?' returns #t if 2 arguments refer to the same object (in memory),  
;; #f otherwise.  
;; In other words, it's a simple pointer comparison.  
(eq? '() '()) ; => #t, since there exists only one empty list in memory  
(let ([x '()] [y '()])  
 (eq? x y)) ; => #t, same as above  
  
(eq? (list 3) (list 3)) ; => #f  
(let ([x (list 3)] [y (list 3)])  
 (eq? x y)) ; => #f — not the same list in memory!  
  
(let\* ([x (list 3)] [y x])  
 (eq? x y)) ; => #t, since x and y now point to the same stuff  
  
(eq? 'yes 'yes) ; => #t  
(eq? 'yes 'no) ; => #f  
  
(eq? 3 3) ; => #t — be careful here  
 ; It’s better to use `=' for number comparisons.  
(eq? 3 3.0) ; => #f  
  
(eq? (expt 2 100) (expt 2 100)) ; => #f  
(eq? (integer->char 955) (integer->char 955)) ; => #f  
  
(eq? (string-append "foo" "bar") (string-append "foo" "bar")) ; => #f  
  
;; `eqv?' supports the comparison of number and character datatypes.  
;; for other datatypes, `eqv?' and `eq?' return the same result.  
(eqv? 3 3.0) ; => #f  
(eqv? (expt 2 100) (expt 2 100)) ; => #t  
(eqv? (integer->char 955) (integer->char 955)) ; => #t  
  
(eqv? (string-append "foo" "bar") (string-append "foo" "bar")) ; => #f  
  
;; `equal?' supports the comparison of the following datatypes:  
;; strings, byte strings, pairs, mutable pairs, vectors, boxes,  
;; hash tables, and inspectable structures.  
;; for other datatypes, `equal?' and `eqv?' return the same result.  
(equal? 3 3.0) ; => #f  
(equal? (string-append "foo" "bar") (string-append "foo" "bar")) ; => #t  
(equal? (list 3) (list 3)) ; => #t  
  
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;  
;; 5. Control Flow  
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;  
  
;;; Conditionals  
  
(if #t ; test expression  
 "this is true" ; then expression  
 "this is false") ; else expression  
; => "this is true"  
  
;; In conditionals, all non-#f values are treated as true  
(member 'Groucho '(Harpo Groucho Zeppo)) ; => '(Groucho Zeppo)  
(if (member 'Groucho '(Harpo Groucho Zeppo))  
 'yep  
 'nope)  
; => 'yep  
  
;; `cond' chains a series of tests to select a result  
(cond [(> 2 2) (error "wrong!")]  
 [(< 2 2) (error "wrong again!")]  
 [else 'ok]) ; => 'ok  
  
;;; Pattern Matching  
  
(define (fizzbuzz? n)  
 (match (list (remainder n 3) (remainder n 5))  
 [(list 0 0) 'fizzbuzz]  
 [(list 0 \_) 'fizz]  
 [(list \_ 0) 'buzz]  
 [\_ #f]))  
  
(fizzbuzz? 15) ; => 'fizzbuzz  
(fizzbuzz? 37) ; => #f  
  
;;; Loops  
  
;; Looping can be done through (tail-) recursion  
(define (loop i)  
 (when (< i 10)  
 (printf "i=~a\n" i)  
 (loop (add1 i))))  
(loop 5) ; => i=5, i=6, ...  
  
;; Similarly, with a named let  
(let loop ((i 0))  
 (when (< i 10)  
 (printf "i=~a\n" i)  
 (loop (add1 i)))) ; => i=0, i=1, ...  
  
;; See below how to add a new `loop' form, but Racket already has a very  
;; flexible `for' form for loops:  
(for ([i 10])  
 (printf "i=~a\n" i)) ; => i=0, i=1, ...  
(for ([i (in-range 5 10)])  
 (printf "i=~a\n" i)) ; => i=5, i=6, ...  
  
;;; Iteration Over Other Sequences  
;; `for' allows iteration over many other kinds of sequences:  
;; lists, vectors, strings, sets, hash tables, etc...  
  
(for ([i (in-list '(l i s t))])  
 (displayln i))  
  
(for ([i (in-vector #(v e c t o r))])  
 (displayln i))  
  
(for ([i (in-string "string")])  
 (displayln i))  
  
(for ([i (in-set (set 'x 'y 'z))])  
 (displayln i))  
  
(for ([(k v) (in-hash (hash 'a 1 'b 2 'c 3 ))])  
 (printf "key:~a value:~a\n" k v))  
  
;;; More Complex Iterations  
  
;; Parallel scan of multiple sequences (stops on shortest)  
(for ([i 10] [j '(x y z)]) (printf "~a:~a\n" i j))  
; => 0:x 1:y 2:z  
  
;; Nested loops  
(for\* ([i 2] [j '(x y z)]) (printf "~a:~a\n" i j))  
; => 0:x, 0:y, 0:z, 1:x, 1:y, 1:z  
  
;; Conditions  
(for ([i 1000]  
 #:when (> i 5)  
 #:unless (odd? i)  
 #:break (> i 10))  
 (printf "i=~a\n" i))  
; => i=6, i=8, i=10  
  
;;; Comprehensions  
;; Very similar to `for' loops -- just collect the results  
  
(for/list ([i '(1 2 3)])  
 (add1 i)) ; => '(2 3 4)  
  
(for/list ([i '(1 2 3)] #:when (even? i))  
 i) ; => '(2)  
  
(for/list ([i 10] [j '(x y z)])  
 (list i j)) ; => '((0 x) (1 y) (2 z))  
  
(for/list ([i 1000] #:when (> i 5) #:unless (odd? i) #:break (> i 10))  
 i) ; => '(6 8 10)  
  
(for/hash ([i '(1 2 3)])  
 (values i (number->string i)))  
; => '#hash((1 . "1") (2 . "2") (3 . "3"))  
  
;; There are many kinds of other built-in ways to collect loop values:  
(for/sum ([i 10]) (\* i i)) ; => 285  
(for/product ([i (in-range 1 11)]) (\* i i)) ; => 13168189440000  
(for/and ([i 10] [j (in-range 10 20)]) (< i j)) ; => #t  
(for/or ([i 10] [j (in-range 0 20 2)]) (= i j)) ; => #t  
;; And to use any arbitrary combination, use `for/fold'  
(for/fold ([sum 0]) ([i '(1 2 3 4)]) (+ sum i)) ; => 10  
;; (This can often replace common imperative loops)