Racket is a descendent of Scheme, which is a descendent of one of the oldest high-level languages, Lisp.\footnote{Lisp was created by AI pioneer John McCarthy in 1959. Notable older high-level languages are Fortran (1954) and Flow-Matic (1955), which was developed by Vassar alumna and professor Grace Hopper. She used her experience with Flo-Matic to help design Cobol.} This family of languages is popular in teaching computer science because of its simplicity. To be only slightly unfair, compare:

*Scheme:*

"Hello world"

*Naming no names:*

```java
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello world");
    }
}
```

While Lisp can be simple, it's also elegant and powerful; Lisp and its derivatives have been fundamental for research in artificial intelligence.

**LANGUAGE BASICS**

*Atomic* elements include:

- *Numbers* like 5 or -3.2
- *Literals* – symbols – like `a`
- *Constants*, including `#t` for true and `#f` for false.

*Non-atomic* elements include:

- *Lists*, the fundamental data structure in any derivative of Lisp, the *list-processing language*. A list is defined recursively: It has the first element and the rest, which is also a list. You’ve reached the end of the list when the rest is the empty list.
- *Pairs* (also called *dotted pairs*), a special kind of list where the second element (the rest) is not a list but another atom. They are represented with a dot, e.g., `(2 . 5).`
LISTS

Lists can contain any kind of value, including other lists, and they can contain any number of values. If a list has zero values, it's called the empty list, (list) or '().

List structure

- **first** or **car**: Return the first element of a given list, e.g.,
  \[(first '(a b)) → 'a\]

- **rest** or **cdr**: Return the list of all elements except the first, e.g.,
  \[(rest '(a b)) → '(b)\]

The names **first** and **rest** are clearer, but **car** and **cdr** are historical. In the original Lisp interpreter (1958), **car** was an abbreviation for “contents or the address part of the register” and **cdr** for “contents of the decrement part of the register” – terminology specific to the IBM 704 the interpreter ran on. Their original meaning is lost, but it wouldn’t be a Lisp language without **car** and **cdr**.

List operators

- **list?**: Test is something is a list.
- **empty?** or **null?**: Test if something is an empty list.
- **length**: Get the number of elements in a list.
- **cons**: Construct a new list consisting of the specified element followed by the list, e.g.,
  \[(cons 'x '(a b)) → '(x a b)\]
  \[(cons '(a b) '(c d e)) → '((a b) c d e)\]

  This makes a new dotted-pair when the second argument is not a list, e.g.,
  \[(cons 'a 'b) → '(a . b)\]

  **Note**: This does not modify the original list you specify. So, if **listy** is a list containing ‘(a b c) and we do (cons ‘x listy), listy still only contains (a b c).

- **append**: Returns a list consisting of the contents of one list appended to the contents of another, e.g.,
  \[(append '(a b) '(c)) → '(a b c)\]

  Also look at **reverse**, **list-ref**, **last**, **member**, **range**.

Using lists for data

When they consist of data, they’re generally quoted, e.g., (quote Bob 32) or, more usually, as the idiomatic form ‘(Bob 32). This prevents Scheme from trying to evaluate the data, e.g., looking for a function Bob.
Lists of data can also begin with the `list` function, which returns a list composed of the arguments. This is different than using `quote` because the arguments will be evaluated before they are put in the new list. Numbers, strings, and booleans are self-quoting, but other values like symbols and expressions that are arguments to `list` will be evaluated unless you quote them. Compare:

```
(quote a b c) → (list 'a 'b 'c)
(list a b c) → error: 'a' undefined
```

Advanced: Look up *quasiquoting* with the backtick operator.

*Using lists for functions*

Lists are also used both to define a function and when we apply a function, e.g., `(+ 5 2)` or `(cons 'a '(b c))`.

*Creating and calling functions*

- `(define name (lambda args) body)`: Defines a function:

  ```racket
  (define my-square (lambda (x) (* x x)))
  (my-square 5) → 25
  ```

- `lambda`: Defines an anonymous function.

  ```racket
  ((lambda (x) (* x x)) 5) → 25
  ```

- `apply`: Applies a function to the arguments found in a list.

  ```racket
  (apply + '(3 3)) → 6
  ```

- `map`: Apply a function to each top-level element in a list, one by one, and return a list of the resulting values. If the function accepts multiple values, multiple lists of the same length are provided:

  ```racket
  (map odd? '(1 2 3)) → (#t #f #t)
  (map (lambda (x) (* 2 x)) '(1 2)) → '(2 4)
  (map + '(1 2 3 4) '(10 20 30 40)) → '(11 22 33 44)
  ```

- Particular to Racket, there are also:

  ```racket
  (andmap odd? '(1 2 3)) = (and (odd? 1) (odd? 2) (odd? 3))
  (ormap even? '(1 2 3)) = (or (even? 1) (even? 2) (even? 3))
  ```

- `filter`: Return only the values from a list that match a predicate, e.g.,

  ```racket
  (filter odd? '(1 2 3 4 5 6 7 8 9)) → '(1 3 5 7 9)
  ```
BRANCHES

It's good style to choose the most specific conditional:

- **when** or **unless** when there's only one branch, e.g.,

  (when (> x 3)
   (print "That's too big. "))
  (unless (list? x)
   (print "Expected a list! "))

- **if**: For at most two branches, e.g.,

  (if (= x 3)
   (print "x is 3")
   (print "x is not 3!"))

- **cond**: For multiple branches. Prefer this over nesting if statements; it's easier to read.

  (cond ((test1) (form1))
       ((test2) (form2))
       ...
       ((testn) (formn)))

  The conditions are evaluated sequentially. The default, “otherwise” condition is else or #t, since it's always true. e.g.,

  ;; Sign function: Return -1 if x is negative, 1 if positive, and 0 otherwise.
  (define sign
   (lambda (x)
     (cond ((< x 0) -1)
           ((> x 0) 1)
           (else 0)))))

- You don't need a conditional when the results are boolean:

  (if (and (number? x) (> 0 x)) #t #f)

  is redundant; just say (and (number? x) (> 0 x)).

  See also: case.

CODING STYLE

- Identifiers – for functions or data – are conventionally written lowercased–with–hyphens. You should follow the conventions of the language you're using; saved identifiers in CamelCase or lower_with_underscores for those other languages.

- Generally, avoid global variables. To declare local variables, use let:
(let ((var1 val1)
       (var2 val2)
       ...
       (varn valn))
  <body using those variables>)

If one of the variable definitions needs to reference one of the variables defined above it, you need to use `let*`.

- If you do need to use a global variable, the convention is to surround the name with asterisks (`*global-var*`), and to use pluses for global constants (`+pi+`).
- As a matter of style for this class, use functional programming as much as possible; avoid parameters and side effects.
- Programming in Scheme often lends itself more gracefully to recursion than iteration:

  ```scheme
  (define incr
    (lambda (x)
      (cond ((null? x)
              '())
            (else
             (cons (+ 1 (car x))
                   (incr (cdr x))))))
  
  (incr '(1 2 3)) → (2 3 4)
  
  But be aware this can have performance trade-offs.
  ```

### COMMON PREDICATES

- `equal?`: Determines equality between the ordered contents of two given lists or strings. It also works on numbers and characters. Prefer this over `eq?`, `eqv?`, and `=` unless you know that you want to use one of those or speed is a concern.
- `null?`: Determines whether a given argument is the empty list.
- `and`, `or`, `not`: Logical predicates. Note that `or` isn't exclusive-or.
- `even?`, `odd?`, `>`, `<`, `>=`, `<=`: Numeric predicates.

### Conjunction and Disjunction

When you use `and` and `or`, Scheme will stop evaluating arguments early if it can:

- If an argument to `and` is found that's false, it stops.
- If an argument to `or` is found that's true, it stops.
**Example Problem: Transforming Lists**

```
(define office-data '(('Mary 221) (John 308) (Jane 221)))
```

Given a list of people’s offices, above, let’s get a list of the names of everyone who has an office. Since a person’s name is the first element of each “entry” sublist, we need to apply the `first` function to each list element in turn and return the result:

```scheme
;; Returns the list of people’s names included in the given office list.
(define get-names
  (lambda (officelist)
    (map first officelist)))

(get-names office-data) → '(Mary John Jane)
```

What if we want to get the list of floors everyone is on? Well, the floor is the first digit of the office number, so we break the problem down: We write a function `get-floor` that turns an office number into a floor number, and we apply it to each element of the office list:

```scheme
;; Given an entry with the name and office number, returns the name and
;; the floor where the office is.
(define get-floor
  (lambda (entry)
    (list (first entry) (floor (/ (second entry) 100)))))

;; Find out the floors on which people are located.
(define get-floors-wrapper
  (lambda (officelist)
    (map get-floor officelist)))

Since we’re only going to use `get-floor` when we’re using `get-floors-wrapper`, we don’t need two separate functions. We can “inline” it by using an unnamed lambda function:

```scheme
;; Find out the floors on which people are located.
(define get-floors
  (lambda (officelist)
    (map (lambda (entry)
            (list (first entry) (floor (/ (second entry) 100))))
         officelist)))

(get-floors office-data) → '(('Mary 2) (John 3) (Jane 2))
```

And if we just want a list of all the floors people are on?

```scheme
(define get-floors-a
  (lambda (officelist)
    (remove-duplicates
     (map second (get-floors officelist)))))

(get-floors-a office-data) → '(2 3)
```
EXERCISES

Always include test cases!

1. Write a function to convert Fahrenheit to Celsius. The formula for this is usually given as $c = \frac{5}{9}(f - 32)$, where $f$ is the temperature in deg. F. and $c$ is the temperature in deg. C.

2. Define a recursive function, `every-other-one`, that takes a list as its only input and returns a list containing “every other” element of the list.

3. Define a function called `toss-die` that takes no input and returns a randomly generated value between 1 and 6.
   - Define a function `sum-n-die-tosses` that takes a single input $n$. It should return as its output the sum of $n$ tosses of a six-sided die.
   - Define a function, called `avg-n-die-tosses`, that takes a single input $n$ and returns as its output the average of $n$ tosses of a six-sided die.

ACKNOWLEDGMENTS

This tutorial adapts descriptions and examples from Matthew Butterick (Beautiful Racket), Luke Hunsberger, Peter Norvig, George Springer & Daniel Friedman (Scheme and the Art of Computer Programming), and Jennifer Walter.
EXERCISE SOLUTIONS

1 (define fahr-to-cels
   (lambda (f)
      (* 5/9 (- f 32.0))))
(fahr-to-cels 212) → 100.0
(fahr-to-cels 32) → 0.0
(fahr-to-cels -40) → -40.0
(fahr-to-cels 0) → -17.77777777777778

2 (define every-other-one
   (lambda (l)
      (cond ((or (null? l) (null? (rest l)))
             l)
            (else
             (cons (first l)
                  (every-other-one (rest (rest l))))))))
(every-other-one '(a b c d e f g h)) → '(a c e g)
(every-other-one '(a b c d e f g h i)) → '(a c e g i)
(every-other-one '(1)) → '(1)
(every-other-one '()) → '()

3 (define toss-die
   (lambda ()
      (+ 1 (random 6))))

(define sum-n-die-tosses
   (lambda (n)
      (if (<= n 0)
          0
          (+ (toss-die)
              (sum-n-die-tosses (- n 1))))))

(define avg-n-die-tosses
   (lambda (n)
      (/ (sum-n-die-tosses n) n 1.0)))