2.2

* Booleans are #t for true and #f for false
  + in conditionals, all non-#f values are true
* Definitions have the form:
  + ( define ‹id› ‹expr› )
    - id is identifiers (functions)
    - expr is the function’s body expressions
* The syntax of a function call is:
  + ( ‹id› ‹expr›\* )
    - ‹id› is the function name
    - ‹expr› is the number of arguments in the function
* The syntax of if conditional is:
  + ( if ‹expr› ‹expr› ‹expr› )
    - the first ‹expr› is always evaluated
      * if it produces a non-#f value, then the second ‹expr› is evaluated for the result of the whole if expression, otherwise the third ‹expr› is evaluated for the result
    - (if (> 2 3)

      "bigger"

      "smaller")

* + - (define (reply s)

  (if (equal? "hello" (substring s 0 5))

      "hi!"

      "huh?"))

* and & or forms make nested conditional easier to read:
  + and shortcuts; it stops and returns #f when an expression produces #f, otherwise it keeps going
  + or similarly short-circuits when it encounters a true result.
  + (define (reply s)

  (if (and (string? s)

           (>= (string-length s) 5)

           (equal? "hello" (substring s 0 5)))

      "hi!"

      "huh?"))

* Nested ifs can also work with individual test sequences
  + (define (reply-more s)

  (if (equal? "hello" (substring s 0 5))

      "hi!"

      (if (equal? "goodbye" (substring s 0 7))

          "bye!"

          (if (equal? "?" (substring s (- (string-length s) 1)))

              "I don't know"

              "huh?"))))

* The syntax for conditionals is:
  + ( cond {[ ‹expr› ‹expr›\* ]}\* )
    - the first ‹expr› is a test expression. If it produces true, then the clause’s remaining ‹expr›s are evaluated, and the last one in the clause provides the answer for the entire cond expression; the rest of the clauses are ignored.
    - If the test ‹expr› produces #f, then the clause’s remaining ‹expr›s are ignored, and evaluation continues with the next clause. The last clause can use else as a synonym for a #t test expression.
  + (define (reply-more s)

  (cond

   [(equal? "hello" (substring s 0 5))

    "hi!"]

   [(equal? "goodbye" (substring s 0 7))

    "bye!"]

   [(equal? "?" (substring s (- (string-length s) 1)))

    "I don't know"]

   [else "huh?"]))

* The actual syntax for function calls is:
  + ( ‹expr› ‹expr›\* )
    - the first ‹expr› is often an ‹id› (like string-append or +), but it can be anything that evaluates to a function
  + (define (double v)

  ((if (string? v) string-append +) v v))

* The syntax for lambda expressions is:
  + ( lambda ( ‹id›\* ) ‹expr›+ )
    - you can use a lambda expression to produce a function directly, especially if you're only going to use it once inside another function
  + (define (twice f v)

  (f (f v)))

* + (twice (lambda (s) (string-append s "!"))

         "hello") vs a separate function being created that does the string-append

* + (define (make-add-suffix s2)

  (lambda (s) (string-append s s2)))

* + > (twice (make-add-suffix "!") "hello")
  + Racket is a lexically scoped language, which means that s2 in the function returned by make-add-suffix always refers to the argument for the call that created the function.
    - the lambda-generated function “remembers” the right s2:
      * (define louder (make-add-suffix "!"))
        + > (twice louder "really")
  + (define (louder s)

  (string-append s "!"))

* + is same as
  + (define louder

  (lambda (s)

    (string-append s "!")))

* Definitions can appear before the body expressions and they're local to the function body:
  + ( define ( ‹id› ‹id›\* ) ‹definition›\* ‹expr›+ )
  + ( lambda ( ‹id›\* ) ‹definition›\* ‹expr›+ )
  + basically nested define statements, with each being local to the level above it (???- Racket example)
  + (define (converse s)

  (define (starts? s2) ; local to converse

    (define len2 (string-length s2))  ; local to starts?

    (and (>= (string-length s) len2)

         (equal? s2 (substring s 0 len2))))

  (cond

   [(starts? "hello") "hi!"]

   [(starts? "goodbye") "bye!"]

   [else "huh?"]))

* Can also create local bindings with the let form:
  + ( let ( {[ ‹id› ‹expr› ]}\* ) ‹expr›+ )
  + each binding clause is an ‹id› and an ‹expr› surrounded by square brackets, and the expressions after the clauses are the body of the let
  + binds many identifiers at once, instead of requiring a separate define for each identifier
  + (let ([x (random 4)]

        [o (random 4)])

    (cond

     [(> x o) "X wins"]

     [(> o x) "O wins"]

     [else "cat's game"]))

* The let\* form, in contrast, allows later clauses to use earlier bindings:
  + (let\* ([x (random 4)]

         [o (random 4)]

         [diff (number->string (abs (- x o)))])

    (cond

     [(> x o) (string-append "X wins by " diff)]

     [(> o x) (string-append "O wins by " diff)]

     [else "cat's game"]))

* + can use earlier identifiers inside later expressions

2.3

* Booleans are #t for true and #f for false
  + in conditionals, all non-#f values are true
* Definitions have the form:
  + ( define ‹id› ‹expr› )
    - id is identifiers (functions)
    - expr is the function’s body expressions
* The syntax of a function call is:
  + ( ‹id› ‹expr›\* )
    - ‹id› is the function name
    - ‹expr› is the number of arguments in the function
* The syntax of if conditional is:
  + ( if ‹expr› ‹expr› ‹expr› )
    - the first ‹expr› is always evaluated
      * if it produces a non-#f value, then the second ‹expr› is evaluated for the result of the whole if expression, otherwise the third ‹expr› is evaluated for the result
    - (if (> 2 3)

      "bigger"

      "smaller")

* + - (define (reply s)

  (if (equal? "hello" (substring s 0 5))

      "hi!"

      "huh?"))

* and & or forms make nested conditional easier to read:
  + and shortcuts; it stops and returns #f when an expression produces #f, otherwise it keeps going
  + or similarly short-circuits when it encounters a true result.
  + (define (reply s)

  (if (and (string? s)

           (>= (string-length s) 5)

           (equal? "hello" (substring s 0 5)))

      "hi!"

      "huh?"))

* Nested ifs can also work with individual test sequences
  + (define (reply-more s)

  (if (equal? "hello" (substring s 0 5))

      "hi!"

      (if (equal? "goodbye" (substring s 0 7))

          "bye!"

          (if (equal? "?" (substring s (- (string-length s) 1)))

              "I don't know"

              "huh?"))))

* The syntax for conditionals is:
  + ( cond {[ ‹expr› ‹expr›\* ]}\* )
    - the first ‹expr› is a test expression. If it produces true, then the clause’s remaining ‹expr›s are evaluated, and the last one in the clause provides the answer for the entire cond expression; the rest of the clauses are ignored.
    - If the test ‹expr› produces #f, then the clause’s remaining ‹expr›s are ignored, and evaluation continues with the next clause. The last clause can use else as a synonym for a #t test expression.
  + (define (reply-more s)

  (cond

   [(equal? "hello" (substring s 0 5))

    "hi!"]

   [(equal? "goodbye" (substring s 0 7))

    "bye!"]

   [(equal? "?" (substring s (- (string-length s) 1)))

    "I don't know"]

   [else "huh?"]))

* The actual syntax for function calls is:
  + ( ‹expr› ‹expr›\* )
    - the first ‹expr› is often an ‹id› (like string-append or +), but it can be anything that evaluates to a function
  + (define (double v)

  ((if (string? v) string-append +) v v))

* The syntax for lambda expressions is:
  + ( lambda ( ‹id›\* ) ‹expr›+ )
    - you can use a lambda expression to produce a function directly, especially if you're only going to use it once inside another function
  + (define (twice f v)

  (f (f v)))

* + (twice (lambda (s) (string-append s "!"))

         "hello") vs a separate function being created that does the string-append

* + (define (make-add-suffix s2)

  (lambda (s) (string-append s s2)))

* + > (twice (make-add-suffix "!") "hello")
  + Racket is a lexically scoped language, which means that s2 in the function returned by make-add-suffix always refers to the argument for the call that created the function.
    - the lambda-generated function “remembers” the right s2:
      * (define louder (make-add-suffix "!"))
        + > (twice louder "really")
  + (define (louder s)

  (string-append s "!"))

* + is same as
  + (define louder

  (lambda (s)

    (string-append s "!")))

* Definitions can appear before the body expressions and they're local to the function body:
  + ( define ( ‹id› ‹id›\* ) ‹definition›\* ‹expr›+ )
  + ( lambda ( ‹id›\* ) ‹definition›\* ‹expr›+ )
  + basically nested define statements, with each being local to the level above it (???- Racket example)
  + (define (converse s)

  (define (starts? s2) ; local to converse

    (define len2 (string-length s2))  ; local to starts?

    (and (>= (string-length s) len2)

         (equal? s2 (substring s 0 len2))))

  (cond

   [(starts? "hello") "hi!"]

   [(starts? "goodbye") "bye!"]

   [else "huh?"]))

* Can also create local bindings with the let form:
  + ( let ( {[ ‹id› ‹expr› ]}\* ) ‹expr›+ )
  + each binding clause is an ‹id› and an ‹expr› surrounded by square brackets, and the expressions after the clauses are the body of the let
  + binds many identifiers at once, instead of requiring a separate define for each identifier
  + (let ([x (random 4)]

        [o (random 4)])

    (cond

     [(> x o) "X wins"]

     [(> o x) "O wins"]

     [else "cat's game"]))

* The let\* form, in contrast, allows later clauses to use earlier bindings:
  + (let\* ([x (random 4)]

         [o (random 4)]

         [diff (number->string (abs (- x o)))])

    (cond

     [(> x o) (string-append "X wins by " diff)]

     [(> o x) (string-append "O wins by " diff)]

     [else "cat's game"]))

* + can use earlier identifiers inside later expressions

2.4

* The cons function actually accepts any two values, but when the second value isn't empty or a list, the result prints in a special way
  + (cons 1 2)
    - returns '(1 . 2)
* cons? = pair?
* first = car
  + (car (cons 1 2))
    - returns 1
* rest = cdr
* The first element of cons can be anything, but the second must be a list to avoid the special notation
  + (cons 1 (list 2 3))
    - returns '(1 2 3)
* (cons 0 (cons 1 2))
  + returns '(0 1 . 2)
* Use the dot notation unless the dot is immediately followed by an open parenthesis. In that case, remove the dot, the open parenthesis, and the matching close parenthesis
  + '(1 . (2 . (3 . ()))) --> '(1 2 3)
* quote lets you write a list as an expression that’s the same as when it prints
  + can use quote or '
  + has no effect on single numbers and strings
    - (quote 42)
      * returns 42
  + (quote ("red" "green" "blue"))
    - returns '("red" "green" "blue")
* Symbol is a value that prints like a quoted identifier
  + symbols have nothing to do with functions and are just made up of the same letters
  + the only difference between symbols and strings is how they're printed
  + (quote jane-doe)
    - returns 'jane-doe
* symbol->string and string->symbol convert between strings and symbols
  + (string->symbol "map")
    - returns 'map
  + (symbol->string (quote map))
    - returns "map"
* The syntax of Racket is determined by two layers:
  + a reader layer, which turns a sequence of characters into lists, symbols, and other constants; and
  + an expander layer, which processes the lists, symbols, and other constants to parse them as an expression.
* The rules for printing and reading go together.
  + e.g. a list is printed with parentheses, and reading a pair of parentheses produces a list

Unix

* A compiler does the following:
  + input: computer program source code
  + Step 1: lexical analysis: convert string of characters to tokens
    - e.g. def add1(n)

return n+1

* + becomes the string of tokens
  + def, add1, leftparen, n, rightparen, return, n, plus, 1
  + Step 2: Parsing: the tokens are parsed into a structure, like a tree
  + Step 3: Code Generation: the parse tree is traversed, and machine code is generated at each node of the tree
  + Output: Machine code, which can be executed on the target machine (Python and Java have increased portability because they generate byte code which is run using a virtual machine)
* UNIX Principles:
  + Principle 1: the file system is a tree
    - directories and subdirectories
* UNIX commands
  + . is the identity directory (every directory has a . that points to itself)
    - cd . leaves you in your current directory
  + .. is the parent directory
    - cd .. brings you to the parent directory
  + cd - takes you back to where you were before
  + ls  gives you information about the directory (when it was last modified, owner, file names, etc)
  + create directories with mkdir name
    - directories in UNIX show up with drwxrwsr-x
  + ls .. gives you information about the parent directory
  + ls .. > xxx prints out the information to an xxx file
    - rewrites output to file
  + cat (concatenation) copies the input to the output
    - can use to read stuff you printed onto file
  + head gives you the first 10 lines of a file
  + tail gives you the last 10 lines of a file
  + mv to rename a file
    - mv xxx yyy renames xxx to yyy
  + diff yyy zzz to print out differences in a file
  + >>
    - appends output to a file
  + date gives you current date and time