CSCI-C311 Programming Languages

Racket: Pairs, Nested Lists, and Vectors

Dr. Hang Dinh

1

Reading Assignment for This Lecture

- The Racket Guide
  - https://docs.racket-lang.org/guide/index.html
  - 2.4 Pairs, Lists, and Racket Syntax
  - 3.9 Vectors
- The Racket Reference
  - https://docs.racket-lang.org/reference/vectors.html
  - 4.12 Vectors

2

Creating Pairs Using The `cons` Function

- Recall: the `cons` function ("construct") adds to the front of the list
- The `cons` function actually accepts any two values
  - not just a list for the second argument.
  - When the second argument is not empty and not itself produced by `cons`, the result is a pair (printed with a dot surrounded by whitespaces):

```
> (cons 1 2)
'(1 . 2)
> (cons "banana" "split")
'("banana" . "split")
```

```
> (cons (list 2 3) 1)
'((2 3) . 1)
> (cons 1 (list 2 3))
'(1 2 3)
```

If you reverse the arguments to cons, you get a non-list pair.

3

Functions `pair?`, `car`, `cdr`

- The function names `cons?`, `rest` make less sense for non-list pairs
  - Function `pair?` is the traditional name for function `cons?`
  - Function `car` is the traditional name for function `first`
  - Function `cdr` (pronounced "could-er") is the traditional name for `rest`

```
> (car (cons 1 2))
1
> (cdr (cons 1 2))
2
> (pair? empty)
#f
> (pair? (cons 1 2))
#t
> (pair? (list 1 2 3))
#t
```

4

The Use of Non-List Pairs

- Non-list pairs are used intentionally, sometimes.
  - The `make-hash` function takes a list of pairs, where the `car` of each pair is a key and the `cdr` is an arbitrary value.
- Printing convention for pairs where the second element is a non-list pair:

```
> (cons 0 (cons 1 2))
'(0 1 . 2)
```

- General rule for printing a pair:
  - Use the dot notation if the dot is not immediately followed by an (
  - Otherwise, remove the dot, the (, and the matching)

`'(0 . (1 . 2))` becomes `'(0 1 . 2)`

`'(1 . (2 . (3 . ())))` becomes `'(1 2 3)`

5

Nested Lists and the `quote` Form

- A list prints with a quote mark before it, but if an element of a list is itself a list, then no quote mark is printed for the inner list:

```
> (list (list 1) (list 2 3) (list 4))
'((1) (2 3) (4))
```

- For nested lists, especially, the `quote` form lets you write a list as an expression in essentially the same way that the list prints:

```
> (quote ("red" "green" "blue"))
'("red" "green" "blue")
> (quote ((1) (2 3) (4)))
'((1) (2 3) (4))
```

6

Nested Lists and the `quote` Form

- The `quote` form also works with the dot notation, whether the quoted form is normalized by the dot-parenthesis elimination rule or not:

```
> (quote (1 . 2))
'(1 . 2)
> (quote (0 . (1 . 2)))
'(0 1 . 2)
```

- Naturally, lists of any kind can be nested:

```
> (list (list 1 2 3) 5 (list "a" "b" "c"))
'((1 2 3) 5 ("a" "b" "c"))
> (quote ((1 2 3) 5 ("a" "b" "c")))
'((1 2 3) 5 ("a" "b" "c"))
```

7

The `quote` Form vs. `list`

- The **`quote`** form doesn't evaluate its arguments, but **`list`** does

```
> (list 1 3 (+ 2 3))
'(1 3 5)
> (quote (1 3 (+ 2 3)))
'(1 3 (+ 2 3))
> (list (circle 10) (circle 20))
'(○ ◯)
> (quote ((circle 10) (circle 20)))
'((circle 10) (circle 20))
```

8

Quoting Pairs and Symbols with `quote`

- If you wrap an identifier with `quote`, then you get output that looks like an identifier, but with a ' prefix:

```
> (quote jane-doe)
'jane-doe
```

```
> map
#<procedure:map>
> (quote map)
'map
```

- A value that prints like a quoted identifier is a *symbol*.
  - A printed symbol should not be confused with an identifier
  - Example: the symbol `(quote map)` has nothing to do with the `map` identifier or the predefined function `map`

9

Symbols v.s. Strings

- Symbols and strings are almost the same thing
  - The intrinsic value of a symbol is nothing more than its character content.
- Main difference is how they print.
- Functions `symbol->string` and `string->symbol` convert between them.

```
> (symbol? (quote map))
#t
> (symbol? map)
#f
> (procedure? map)
#t
> (string->symbol "map")
'map
> (symbol->string (quote map))
"map"
```

10

Quoting Pairs and Symbols with `quote`

- `quote` for a list automatically applies itself to nested lists
- Similarly, `quote` on a parenthesized sequence of identifiers automatically applies itself to the identifiers to create a list of symbols

```
> (quote (road map))
'(road map)
> (car (quote (road map)))
'road
```

- The quote form has no effect on a literal expression such as a number or string:

```
> (quote 42)
42
> (quote "on the record")
"on the record"
```

11

Vectors

- A *vector* is a fixed-length array of arbitrary values.
  - Unlike a list, a vector supports constant-time access and update of its elements.
- A vector prints similar to a list
  - as a parenthesized sequence of its elements—but is prefixed with `#` after `'`
  - or it uses `vector` if one of its elements cannot be expressed with `quote`.

```
> #("a" "b" "c")
'#("a" "b" "c")
> #(name (that tune))
'#(name (that tune))
```

14

Vectors as Expressions

- A vector as an expression can be supplied with an optional length

```
> #4(baldwin bruce)
'#(baldwin bruce bruce bruce)
```

- A vector as an expression implicitly `quote`s the forms for its content
  - Identifiers and parenthesized forms in a vector constant represent symbols and lists

```
> #(name (1 2))
'#(name (1 2))
```

- Vectors written as expressions are immutable, like literal strings
  - `#` is an alias for **`vector-immutable`**

15

The vector Form vs.

- Vectors created with the vector form are mutable

```
> (vector 1 2 "a" "b")
'#(1 2 "a" "b")
```

- The vector form evaluates its arguments but the # form doesn't

```
> (vector name (1 2))
name: undefined;
 cannot reference an identifier before its definition
> (vector (circle 10) (circle 15) (circle 20))
'#(o o o)
> #((circle 10) (circle 15) (circle 20))
'#((circle 10) (circle 15) (circle 20))
```

16

Useful Vector Functions

- Function `vector-ref` returns an element at a position in a vector

```
> (vector-ref #("a" "b" "c") 1)
"b"
> (vector-ref #(name (that tune)) 1)
'(that tune)
```

- Vectors can be converted to lists and vice versa via `vector->list` and `list->vector`

```
> (list->vector (map string-titlecase
                     (vector->list #("three" "blind" "mice"))))
'#("Three" "Blind" "Mice")
```

17

Useful Vector Functions

- Function make-vector creates a mutable vector of same elements

```
> (make-vector 5 0)
'#(0 0 0 0 0)
> (apply hc-append
         (vector->list (make-vector 10 (circle 20))))
```

- Function vector-append concatenates two vectors

```
> (vector-append #(1 2) #(3 4 5))
'#(1 2 3 4 5)
> (vector-append (vector (circle 10))
                 (make-vector 5 (circle 20)))
'#(
```

)

18