CSCI-C311 Programming Languages

Racket: Simple Values, Identifiers, Conditionals

Dr. Hang Dinh

1

Reading Assignment for This Lecture

- The Racket Guide
  - https://docs.racket-lang.org/guide/index.html
  - Section 2.1 – Simple Values; Section 3.5 - Bytes and Bytes Strings
  - Sections 2.2.1 – 2.2.3
  - Section 2.2.5 - Conditionals with if, and, or, and cond

2

Simple Values in Racket

- Racket values include *numbers*, *booleans*, *strings*, and *byte strings*.
- *Numbers* are written in the usual way,
  - including fractions and imaginary numbers:

```
1        3.14
1/2      6.02e+23
1+2i     9999999999999999999999
```

- *Booleans* are #t for true and #f for false.
  - In conditionals, however, all non-#f values are treated as true.

3

Simple Values in Racket: Strings

- *Strings* are written between doublequotes.
- Within a string, backslash \ is an escaping character

```
"Benjamin \"Bugsy\" Siegel”
```

- Except for an unescaped doublequote or backslash, any Unicode character can appear in a string constant.
  - An Unicode character is a hexadecimal escape with \u (up to four digits)

```
> "Bugs \u0022Figaro\u0022 Bunny"
"Bugs \"Figaro\" Bunny"
```

4

Simple Values in Racket: Bye Strings

- A byte is an integer in between 0 and 255.

```
> (byte? 0)
#t
> (byte? 256)
#f
```

The byte? predicate recognizes numbers that represent bytes.

- A *byte string* is similar to a string but its content is a sequence of bytes instead of characters.

```
> #"Apple"
#"Apple"
> (bytes-ref #"Apple" 0)
65
> (make-bytes 3 65)
#"AAA"
```

A byte string prints like the ASCII decoding of the byte string, but prefixed with a #.

5

Identifiers in Racket

- Racket’s syntax for identifiers is especially liberal.
  - An identifier is any sequence of non-whitespace characters except the special characters `( ) [ ] { } " , ' ` ; # | \`
  - And except for the sequences characters that make number constants
- Examples of identifiers

  substring
  hc-append
  a+b-1
  Pass/Fail?
  +

6

Conditional: `if`

- The `if` conditional expression is of the form

  (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.
- Example:

```
> (if (> 2 3)
      "2 is larger than 3"
      "3 is larger than 2")
"3 is larger than 2"
```

Operators are treated as functions in Racket. So you cannot write (2 > 3) when using operator >.

7

Conditional: `if`

- Complex conditionals can be formed by nesting if expressions

```
(define (reply-non-string s)
  (if (string? s)
      (if (string-prefix? s "hello ")
          "hi!"
          "huh?")
      "huh?"))
```

Predefined function `string?` returns true iff its argument is a string.

Predefined function `string-prefix` returns true iff its 1st argument contains its 2nd argument as a prefix.

8

Conditional: nested `if`

- Instead of duplicating the "huh?" case, the `reply-non-string` function is better written as

```
(define (reply-non-string s)
  (if (if (string? s)
          (string-prefix? s "hello ")
          #f)
      "hi!"
      "huh?"))
```

- But these kinds of nested ifs are difficult to read.

9

Conditionals: `and, or`

- Racket provides more readable shortcuts via the `and` and `or` forms:

  (and ‹expr›*)

  (or ‹expr›*)

  - ‹expr›* in the syntax denotes a sequence of zero or more expressions
  - The and form short-circuits: it stops and returns #f when an expression produces #f, otherwise it keeps going.
  - The or form similarly short-circuits when it encounters a true result.

10

Conditionals: and, or

- Example:

```
(define (reply-non-string s)
  (if (and (string? s) (string-prefix? s "hello "))
      "hi!"
      "huh?"))

> (reply-non-string "hello racket")
"hi!"
> (reply-non-string 17)
"huh?"
```

11

Conditionals: if, cond

- Another common pattern of nested ifs involves a sequence of tests, each with its own result

```
(define (reply-more s)
  (if (string-prefix? s "hello ")
      "hi!"
      (if (string-prefix? s "goodbye ")
          "bye!"
          (if (string-suffix? s "?")
              "I don't know"
              "huh?"))))
```

Predefined function string-suffix returns true iff its 1st argument contains its 2nd argument as a suffix.

12

Conditional: cond

- The shorthand for a sequence of tests is the `cond` form:

  (cond {[‹expr› ‹expr›*]}*)

- A cond form contains a sequence of clauses between [ ] brackets.
- In each clause, the first ‹*expr*› is a test expression.
  - If the test ‹*expr*› produces #f, then the clause's remaining ‹*expr*›s are ignored, and evaluation continues with the next clause.
- If the first ‹*expr*› of a clause produces true, then
  - the clause's remaining ‹*expr*›s are evaluated.
  - the last one in the clause provides the answer for the entire cond expression;
  - the remaining clauses are ignored
- The last clause can use else as a synonym for a #t test expression.

13

Conditional: cond

- Rewrite the `reply-more function`:

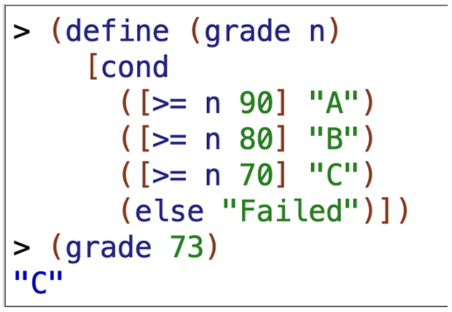
```
(define (reply-more s)
  (cond
   [(string-prefix? s "hello ")
    "hi!"]
   [(string-prefix? s "goodbye ")
    "bye!"]
   [(string-suffix? s "?")
    "I don't know"]
   [else "huh?"]))
```

```
> (reply-more "hello racket")
"hi!"
> (reply-more "goodbye cruel world")
"bye!"
> (reply-more "what is your favorite color?")
"I don't know"
> (reply-more "mine is lime green")
"huh?"
```

14

The Use of Square Brackets

- The use of square brackets for cond clauses is a convention.
- In Racket, ( ) and [ ] brackets are actually interchangeable,
  - as long as (is matched with) and [is matched with].
  - Using [ ] brackets in a few key places makes Racket code even more readable.



```
> (define (grade n)
    [cond
      ([>= n 90] "A")
      ([>= n 80] "B")
      ([>= n 70] "C")
      (else "Failed")])
> (grade 73)
"C"
```

15