CSCI-C311 Programming Languages

Racket: Regular Expressions

Dr. Hang Dinh

1

Reading Assignment for This Lecture

- The Racket Guide
  - Part 9. Regular Expressions
    - 9.1 Writing Regexp patterns
    - 9.2 Matching Regexp patterns
    - 9.3 Basic Assertions
    - 9.4 Characters and Character Classes
    - 9.5 Quantifiers
    - 9.6 Clusters
    - 9.7 Alternation
- The Racket Reference
  - 4.8 Regular Expressions

2

Regular Expressions in Racket

- Regular expressions describe **patterns** of strings that can be combined from characters using *concatenation*, *alternation*, and *Kleene closure*.
- Racket supports regular expressions with *regexp* (and *pregexp*) values
  - A *regexp* value encapsulates a pattern described by a string or byte string, using the same pattern language as either the Unix utility *egrep* or Perl.
  - The regexp matcher (e.g., function regexp-match) tries to match this pattern against (a portion of) another string or byte string, called the *text string*.
  - The text string is treated as raw text, and not as a pattern.
  - pregexp values are similar to regexp but specified with slightly extended syntax

3

Writing regexp Values

- To form a literal regexp value, prefix a string or byte string with #rx

```
> #rx"abc" ;a string-based regexp value
#rx"abc"
> #rx#"abc" ;a byte string-based regexp value
#rx#"abc"
```

- Function regexp takes a string and compiles it into a regexp value
  - Use regexp when you construct a pattern to be matched against multiple strings, since a pattern is compiled to a regexp value before it can be used in a match.
- Function regexp? takes an argument and returns true if it's a regexp

```
> (regexp "abc")
#rx"abc"
```

```
> (regexp? "(.*)\\1")
#f
> (regexp "(.*)\\1")
#rx"(.*)\\1"
```

4

Metacharacters in regexp Patterns

- Most of the characters in a regexp pattern are *literals*
  - They are meant to match occurrences of themselves in the text string.
  - Thus, the pattern #rx"abc" matches a string that contains the characters a, b, and c in succession.
- Other characters act as *metacharacters*, and some character sequences act as *metasequences*.
  - They specify something other than their literal selves.
  - Example: in the pattern #rx"a.c", the metacharacter . can match *any* character.
  - To match the . itself, we escape it with a \. The sequence \. is a metasequence

(Note: we have to use \\ as in #rx"a\\.c“ to escape a metacharacter)

5

Metacharacters in regexp Patterns; Basic Assertions

- More metacharacters in regexp:

^ $ \ ( ) [ ] | * + ?

- The assertion ^ (outside [ ]) specifies the *start* of the text string
  - Pattern #rx”^abc" matches a string that starts with the substring abc
  - Pattern #rx”[^abc]" matches any character other than a b c
- The assertion $ specifies the *end* of the text string
  - Pattern #rx”abc$" matches a string that ends with the substring abc
  - Pattern #rx”$" matches a string that ends with an empty string (so it matches every string)

6

Matching Regexp Patterns Using regexp-match-positions function

- The regexp-match-positions function
  - takes a regexp pattern and a text string,
  - returns a match if the regexp matches (some part of) the text string, or #f otherwise.
  - A successful match produces a list of *index pairs*.
- Example:

```
> (regexp-match-positions #rx"hello" "Hello")
#f
> (regexp-match-positions #rx"are" "How are you?")
'((4 . 7))
> (substring "How are you?" 4 7)
"are"
```

7

Matching Regexp Patterns Using regexp-match-positions function

- regexp-match-positions function's optional 3rd and 4th arguments
  - specify indices of the text string within which the matching should take place.
- Example:

```
> (regexp-match-positions #rx"are"
    "How are you? You are happy, aren't you?" 7 20)
'((17 . 20))
> (regexp-match-positions #rx"are"
    "How are you? You are happy, aren't you?" 19 40)
regexp-match-positions: ending index is out of range
  ending index: 40
> (regexp-match-positions #rx"are"
    "How are you? You are happy, aren't you?" 19 30)
#f
> (regexp-match-positions #rx"are"
    "How are you? You are happy, aren't you?" 19 35)
'((28 . 31))
```

8

Matching Regexp Patterns Using regexp-match function

- The regexp-match function is like regexp-match-positions, but instead of returning index pairs, it returns the matching substrings

```
> (regexp-match #rx"are" "How are you?")
'("are")
> (regexp-match #rx"[^abc]" "bye bye!")
'("y")
> (regexp-match #rx"^How" "How are you?")
'("How")
> (regexp-match #rx"ou?$" "How are you?")
#f
> (regexp-match #rx"ou\?$" "How are you?")
read-syntax: unknown escape sequence `\?` in string
> (regexp-match #rx"ou\\?$" "How are you?")
'("ou?")
```

9

Inverse Matching Regexp Patterns

- The regexp-quote function takes an arbitrary string and returns a string for a pattern that matches exactly the original string.
  - Characters in the input string that could serve as regexp metacharacters are escaped with a backslash, so that they safely match only themselves.

```
> (regexp-quote "What?")
"What\\?"
> (regexp-quote "2^3=8")
"2\\^3=8"
> (regexp-quote "[index]=(n-1)")
"\\[index\\]=\\(n-1\\)"
```

10

Character Classes

- A *character class* matches any one character from a set of characters.
  - A typical format for this is the *bracketed character class* [...], which matches any one character from characters enclosed within the brackets.
  - Thus, #rx"p[aeiou]t" matches pat, pet, pit, pot, put, and nothing else.
- Inside the brackets,
  - a - between 2 characters specifies the Unicode range between the characters.
  - For example, #rx"ta[b-dgn-p]" matches tab, tac, tad, tag, tan, tao, and tap.
  - Most other metacharacters (., *, +, ?, etc) cease to be metacharacters
- A] immediately occurring after [is also not a metacharacter.
  - For example, #rx"[]ab]" matches], a, and b.

11

Quantifiers and Kleene Closure

- The *quantifier*
  - * matches 0 or more
  - + matches 1 or more
  - ? matches 0 or 1

instances of the preceding subpattern.

- The metacharacter * specifies Kleene star.

```
> (regexp-match-positions #rx"c[ad]*r" "cadaddadddr")
'((0 . 11))
> (regexp-match-positions #rx"c[ad]*r" "cr")
'((0 . 2))
> (regexp-match-positions #rx"c[ad]+r" "cadaddadddr")
'((0 . 11))
> (regexp-match-positions #rx"c[ad]+r" "cr")
#f
> (regexp-match-positions #rx"c[ad]?r" "cadaddadddr")
#f
> (regexp-match-positions #rx"c[ad]?r" "cr")
'((0 . 2))
> (regexp-match-positions #rx"c[ad]?r" "car")
'((0 . 3))
```

12

Clusters - Subpatterns

- Clustering—enclosure within parens (...)
  - identifies the enclosed subpattern as a single entity.
  - causes the matcher to capture the *submatch*, or the portion of the text string matching the subpattern, in addition to the overall match

```
> (regexp-match #rx"([a-z]+) ([0-9]+), ([0-9]+)" "jan 1, 1970")
'("jan 1, 1970" "jan" "1" "1970")
```

overall match 3 *submatches*

Number of submatches = Number of subpatterns

  - causes a following quantifier to treat the entire enclosed subpattern as an entity

```
> (regexp-match #rx"t([a-c];)*" "ta;b;c; end")
'("ta;b;c;" "c;")
```

The *-quantifier subpattern matches 3 times but only the last submatch is returned

13

Clusters

- A quantified subpattern may to fail to match, even if the overall pattern matches.
  - In such cases, the failing submatch is represented by #f

```
> (define date-re
    ; match 'month year' or 'month day, year';
    ; subpattern matches day, if present
    #rx"([a-z]+) +([0-9]+,)? *([0-9]+)")
> (regexp-match date-re "jan 1, 1970")
'("jan 1, 1970" "jan" "1," "1970")
> (regexp-match date-re "jan 1970")
'("jan 1970" "jan" #f "1970")
```

14

Alternation

- The | separates *alternate* subpatterns in the nearest enclosing cluster (or in the entire pattern string if there are no enclosing parens).
  - If you wish to use clustering merely to specify a list of alternate subpatterns but do not want the submatch, use (?: instead of (.

```
> (regexp-match #rx"f(ee|i|o|um)" "a small, final fee")
'("fi" "i")
> (regexp-match #rx"f(?:ee|i|o|um)" "a small, final fee")
'("fi")
```

- Note: the leftmost matching alternate is picked

```
> (regexp-match #rx"Hi|Hi there!" "Hi there!")
'("Hi")
```

15