Discussion Session 7: Getting Familiar with Racket

Installation

Racket: https://download.racket-lang.org/
Rosette: https://github.com/emina/rosette#installing-rosette

The Racket Guide

1. **Define and evaluate a function**
   - Let's define a simple function:

```
#lang racket

(define (extract str)
  (substring str 2 5))
```

   - Notes:
     - The build-in function `substring` is defined as:

       ```
       (substring str start [end]) → string?
       ```

     - Returns a new mutable string that is `(- end start)` characters long, and that contains the same characters as `str` from `start` inclusive to `end` exclusive.
   - In the defined function, `substring` takes three arguments: `str 2 5`, which returns the 2th to 4th characters in `str`. The output is of type `string`.
   - Then we want to call function `extract` to get the 2th to 4th characters in `"I can learn Racket well."`
   - Two ways to evaluate your `define` form:
     - In `DrRacket`, you'd normally put the definition in the top text area—called the *definitions area*—along with the `#lang racket`, click **Run** and then put `(extract "I can learn Racket well.")` in the `REPL`.
     - In command-line `racket`, first save the definition in a `*.rkt` file (`e.g. extract.rkt`), then start `racket` in the same directory, and follow the steps below:

       ```
       > (enter! "extract.rkt")
       > (extract "I can learn Racket well.")
       "can"
       ```

   - Note:

- The `enter!` form is just like `DrRacket`'s **Run** button.

2. **Some basic list operations**
   - Define a list using the `list` function, which takes any number of values and returns a list containing the values, e.g.:

```
> (list "r" "a" "c" "k" "e" "t")
'("r" "a" "c" "k" "e" "t")

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

   - Predefined list functions:

```
> (length (list "r" "a" "c" "k" "e" "t")) ; count the elements
6

> (append (list "r" "a" "c") (list "k" "e" "t")) ; combine lists
'("r" "a" "c" "k" "e" "t")

> (reverse (list "r" "a" "c" "k" "e" "t")) ; reverse order
'("t" "e" "k" "c" "a" "r")

> (member "a" (list "r" "a" "c" "k" "e" "t")) ; check for an element.
If such an element exists, the tail of the list starting with that
element is returned.
'("a" "c" "k" "e" "t")

> (member "m" (list "r" "a" "c" "k" "e" "t")) ; check for an element.
Otherwise, the result is #f.
#f
```

   - Predefined list loops:

```
> (map sqrt (list 1 4 9 16)) ; the map function uses the per-element
results to create a new list
'(1 2 3 4)

> (andmap string? (list "a" "b" "c"))
#t

> (andmap string? (list "a" "b" 6))
```

```
#f

> (ormap number? (list "a" "b" 6))
#t

> (filter string? (list "a" "b" 6)) ; the filter function keeps
elements for which the body result is true, and discards elements for
which it is #f
'("a" "b")
```

- Define your own functions, e.g.:

```
(define (my-length lst)
  (cond
   [(empty? lst) 0]
   [else (+ 1 (my-length (cdr lst)))]))

> (my-length empty)
0

> (my-length (list "a" "b" "c"))
3
```

Convert the previous definition to a tail-recursion way:

```
(define (my-length-tail lst)
  ; local function iter:
  (define (iter lst len)
    (if (empty? lst)
        len
        (iter (cdr lst) (+ len 1))))
  ; body of my-length calls iter:
  (iter lst 0))
```

3. More tutorial

Racket: https://docs.racket-lang.org/guide/
Rosette: https://docs.racket-lang.org/rosette-guide/index.html