University of Waterloo
David R. Cheriton
School of Computer Science

Unit 2 – Separation of Concerns + Abstraction

- Separate code which *implements* functionality from code which *uses* functionality.
- How should the appropriate functionality (functions, data) be described?
- An Abstract Data Type (ADT) is a formal description of a collection of data.
- ADTs are an important approach to separating the specification of a type of data from its implementation. They are beneficial both for the design and implementation of algorithms and data structures.

Simon Game

- “Do what Simon says.”
- The machine/leader presents a sequence of colours.
  - It would usually present these quickly.
- The player repeats back the same colours in the same order.
  - Each colour is a button on the game board, so the player must press the colour buttons in the right order.
- Each round of the game adds one new colour to the sequence.

Simon: Example

- Simon says `red`
  - you press red
- Simon says `red yellow`
  - you press red yellow
- Simon says `red yellow yellow`
  - you press red yellow yellow
- ...

A Simple Simon Game

Let's implement a "Simple Simon" game:

- Just one round of play—Simon picks one sequence of length n, and the player must repeat back the sequence.
- We can separate two concerns here:
  - How to build the Simon game logic
    - Includes picking the sequence, checking the keypresses, detecting wins and losses.
  - How to build Simon's opponent (a client)
    - Who is Simon's opponent? (human, computer, something else?)
    - How does the opponent interact with Simon?
- Start by defining the interface between these two concerns.
- What functions do we need, what do they consume, what are their side-effects, what do they produce?

Simple Simon Interface

```
;; A Colour = (union 'red 'blue 'green 'yellow)

;; simon.rkt plays a single round of the "Simon" game
;; provides:
;;   simple-simon:  Nat -> (listof Colour)
;;                 PRE: n >= 1
;;                 POST: produces a list of length n
;;                       game can start
;;   (simple-simon n) produces a sequence of n Colours
;;
;;   press:  Colour -> (union 'win 'ok 'lose)
;;          PRE: simple-simon has been called
;;                game not done
;;          POST: Produces Symbol according to Simon Game Rules
;;                Advances the game one move
;;   (press c) consumes a colour c and produces
;;     'win if c is correct and game done
;;     'ok if correct and game not done
;;     'lose if c is incorrect
```

We'll develop code that implements this interface in class.

Two Different Clients

- We will build two different clients (players) for Simon:
  - A computer player – it narrates its interaction with Simon, and it never loses.
  - A human player – the user is guided through the game and prompted for keypresses.
- Both of these modules conform to the same interface (both provide `play-simple-simon`).
- So we can swap one in for the other if desired.

A Computer Player for Simon

```racket
;; Provides (play-simple-simon n)
;; - Generates simple-simon game of size n
;; - Plays the game, printing presses and results as we go.
#lang racket
(require "simon.rkt")
(provide play-simple-simon)

(define (press-next lst)
  (cond [(empty? lst) (printf "All done!\n") ]
        [else
         (define next-to-press (first lst))
         (printf "Pressing ~a...\n" next-to-press)
         (define press-result (press next-to-press))
         (printf "Result: ~a.\n" press-result)
         (press-next (rest lst))]))

(define (play-simple-simon n)
  (define colour-lst (simple-simon n))
  (printf "Colours to match: ~a\n" colour-lst)
  (press-next colour-lst))
```

A Testing Module

```
#lang racket  ;; simon-computer-player-test.rkt
(require "simon-computer-player.rkt")
(play-simple-simon 3)
(play-simple-simon 5)
(play-simple-simon 10)
```

```
Colours to match: (yellow yellow blue)
Pressing yellow...
Result: ok.
Pressing yellow...
Result: ok.
Pressing blue...
Result: win.
All done!
Colours to match: (blue green red yellow yellow)
.......
```

A Human Player for Simon

```
;; Provides (play-simple-simon n)
;; -Generates simple-simon game of size n
;; -Plays the game, printing presses and results as we go.

#lang racket
(require "simon.rkt")
(provide play-simple-simon)

(define (press-next) (printf "Enter a colour: ")
  (define next-to-press (read))
  (printf "Pressing ~a..." next-to-press)
  (define press-result (press next-to-press))
  (printf "Result: ~a.\n" press-result)
  (cond [(symbol=? press-result 'lose)
         (printf "You lost!!\n")]
        [(symbol=? press-result 'win)
         (printf "You won!!\n")]
        [else (press-next)]))

(define (play-simple-simon n)
  (define colour-lst (simple-simon n))
  (printf "Colours to match: ~a\n" colour-lst)
  (press-next))
```

A Testing Module

```
#lang racket  ;; simon-human-player-test.rkt
(require "simon-human-player.rkt")
(play-simple-simon 3)
(play-simple-simon 5)
(play-simple-simon 10)
```

```
Colours to match: (red blue yellow)
Enter a colour: red
Pressing red...Result: ok.
Enter a colour: blue
Pressing blue...Result: ok.
Enter a colour: green
Pressing green...Result: lose.
You lost!!
Colours to match: (blue green blue blue blue)
.......
```

An Implementation of the Game

```
;; A Colour = (union 'red 'blue 'green 'yellow)

;; simon.rkt plays a single round of the "Simon" game
;; provides:
;;   simple-simon:  Nat -> (listof Colour)
;;                 PRE: n >= 1
;;                 POST: produces a list of length n
;;                       game can start
;;   (simple-simon n) produces a sequence of n Colours
;;
;;   press:  Colour -> (union 'win 'ok 'lose)
;;          PRE: simple-simon has been called
;;                game not done
;;          POST: Produces Symbol according to Simon Game Rules
;;                Advances the game one move
;;   (press c) consumes a colour c and produces
;;     'win if c is correct and game done
;;     'ok if correct and game not done
;;     'lose if c is incorrect
(provide simple-simon press)
```

```
;; random-colour:  -> Colour
;; PRE: true
;; POST: produces a random Colour
(define (random-colour)
  (define r (random 4))
  (cond
    [(= r 0) 'blue]
    [(= r 1) 'red]
    [(= r 2) 'yellow]
    [else 'green]))

;; Mutable variable to hold the colours that
;; need to be pressed to win the game
(define colours empty)
```

```
;; (simple-simon n) provided by module; see interface
(define (simple-simon n)
  ;; make-colour-list:  Nat -> (listof Colour)
  ;; generate a list of x colours
  (define (make-colour-list x)
    (cond
      [(= x 0) empty]
      [else (cons (random-colour)
                  (make-colour-list (sub1 x)))]))
  (set! colours (make-colour-list n))
  colours)

;; (press c) provided by module; see interface
(define (press c)
  (define shouldbe (first colours))
  (set! colours (rest colours))
  (cond
    [(not (equal? c shouldbe)) 'lose]
    [(empty? colours) 'win]
    [else 'ok]))
```

A Fancier User Interface (UI)

- A *user interface* describes the way in which the computer interacts with the user.
- `simon-human-player.rkt` provided a text-based user interface for Simon.
- For a fancier UI, look for `simon-ui.rkt` on the course website.
- This module `require`s a module `keyboard.rkt`
  - Uses system calls and other fancy things
  - Must be run in via `runC` (on linux or Mac)
- You are not responsible for these, but you can take a look
- Note: this is a good example of separation of concerns. The interface is isolated so it can be implemented by means you do not need to understand.

Separation of Concerns

- `simon.rkt` provides the underlying "guts" of the game
- Our computer player code, human player code, and the fancy UI code all use `simon.rkt`
- The underlying implementation could change; no harm done, **as long as the interface stays the same.**
- Separation of concerns: `simon.rkt` handles remembering and checking, other modules handle deciding what to press

Separation of Concerns: Data

- In `simon.rkt`, we separated the code that *implements* the game (the “guts”) from the code that *uses* the game (the player modules).
- If we take the same approach to our data structures, we have the following concerns:
  - how we *represent* (or implement) the data structure;
  - how we *use* the data structure.
- Now, how do we separate them?

Remember this?

- In CS 135, we said things like
  `;; An association list (AL) = (listof (list Num String))`
- This gives away too much information! Now the client knows how an AL is implemented, and even a little knowledge can be dangerous.
- **Do not say** "this is what the data looks like".
  Do say "this is how you use the data".
- In essence: "Never mind how my data is built—I'll give you functions you can call, and if you just stick to those, we'll both be happy."
- So instead, say: "I provide a dictionary structure for name-value lookups. You can use it by calling the functions `new-dict`, `insert`, `lookup`, and `remove`."
- If you do this, you have an **Abstract Data Type (ADT)**.

Abstract Data Types (ADTs)

- An ADT is a formal specification of a collection of data.
- But we specify **not** the data itself, but instead the available operations on the data.
  - Useful in separating specification from implementation
  - One ADT specification can have several implementations
  - Description uses mathematical notation (sets, sequences) **not code**
- The specification of an *operation* for an ADT includes:
  - List and brief **description of parameters**;
  - A **precondition**;
  - A **postcondition**.

ADT Example: Back to the passport office

- Needed a way to keep track of
  - The order in which people came in – gave numbers to "clients"
  - The order in which we serve people – gave numbers to "servers"
- Suppose now that we need to keep track of the actual people waiting, and ensure that people are served in the order in which they arrived.
- We need a data structure with a First-In-First-Out (FIFO) behaviour:
  - The first item we can retrieve from the structure is the first item we put into it.
  - Like a “pipeline” in which you insert at one end and retrieve at the other.
- We have just (informally) specified **ADT Queue**.

ADT Queue: Specification

Conceptually, a queue is a (possibly empty) sequence $(q_1, q_2, ..., q_n)$, in which items are inserted at one end (enqueue) and retrieved at the other (dequeue).

Operations

- new-queue : $\rightarrow$ Queue
  - Takes no parameters (note syntax above for contracts)
  - PRE: True (operation can always be done)
  - POST: Produces an empty queue
- queue-empty? : Queue $\rightarrow$ Boolean
  - One parameter, a queue $Q = (q_1, q_2, ..., q_n)$
  - PRE: True
  - POST: produces True if sequence is empty, False otherwise
- ...

Queue: More interesting operations

- enqueue : Queue Any $\rightarrow$ Queue
  - Two parameters, an item e and a queue $Q = (q_1, q_2, ..., q_n)$
  - PRE: True
  - POST: Produces $Q' = (e, q_1, q_2, ..., q_n)$
- head : Queue $\rightarrow$ Any
  - One parameter, a queue $Q = (q_1, q_2, ..., q_n)$.
  - PRE: $n \geqslant 1$
  - POST: Produces value q_n.
- dequeue : Queue $\rightarrow$ Queue
  - One parameter, a queue $Q = (q_1, q_2, ..., q_n)$
  - PRE: $n \geqslant 1$
  - POST: Produces $Q' = (q_1, ..., q_{n-1})$.

Now, even though we don't know how a queue is built, we know exactly how it should behave.

An obvious Racket implementation

Here is a sample implementation of a queue using a list.

In file `queue.rkt`:

```
#lang racket
(provide new-queue queue-empty?
  enqueue head dequeue)
(define (new-queue) empty)
(define (queue-empty? q)
  (empty? q))
(define (enqueue q item)
  (cons item q))
(define (head q)
  (last q))
(define (dequeue q)
  (drop-right q 1))
```

`drop-right` is a standard Racket function which consumes a list `lst` and a number `k` and produces a list consisting of all items in `lst` except the last `k`.

Queue Use

Testing Code

```
#lang racket  ;; "queue-ex1.rkt"
(require "queue.rkt")
(define Q
  (enqueue
    (enqueue (new-queue) 'tomato)
                             'onion))
;;Postconditions imply n = 2 here




(queue-empty? Q)
(head Q)
(define Q2 (dequeue Q))
(head Q2)
(define Q3 (dequeue Q2))
;;Postconditions imply n = 0 here
(queue-empty? Q3)
```

Output

```
#f
'tomato

'onion


#t
```

Queue Abuse

Testing Code

```
#lang racket  ;; "queue-ex2.rkt"
(require "queue.rkt")
(define Q
  (enqueue
    (enqueue (new-queue) 'tomato)
                             'onion))
;;Postconditions imply n = 2 here

;; I hate tomatoes!!
Q
(set! Q (list (first Q)))

(queue-empty? Q)
(head Q)
(define Q2 (dequeue Q))
(head Q2)
(define Q3 (dequeue Q2))
;;Postconditions imply n = 0 here
(queue-empty? Q3)
```

Output

```
'(onion tomato)

#f
'onion

queue.rkt:7:22: last:
expected argument of
type <non-empty list>;
given '()
```

What's the problem?

- Poorly-kept secret: Our queue is just a list
  - We can tamper with our queues by extracting elements out of order.
- We can make lists easily with built-in Racket functions.
  - Therefore we can "forge" queues and cause unexpected results
- So what?
  - Temptation by ADT clients to cheat.
  - Even honest clients may accidentally cheat.
  - Unexpected behaviour.

Data-hiding: Structs in Racket

Recall from CS135 the `define-struct` special form:

```
(define-struct mytype (field-a field-b ...))
```

- Defines the following functions:
  - Constructor: `(make-mytype val-for-a val-for-b ...)`
    - Makes a new struct, fills in fields with given values
  - Accessors: `(mytype-field-a the-mytype)`, `(mytype-field-b the-mytype)`, ...
    - Takes an existing struct, returns value of implied field
  - Type-checker: `(mytype? the-thing)`
    - `true` if `the-thing` was created by `(make-mytype ...)`, `false` otherwise.
- If we hide the definition in another module, these functions are unuseable unless we `provide` them.
- Our structs cannot be tampered with! Or forged!

Example: Sammich

Example Code

```
#lang racket ;; file "sammich-firsttry.rkt"
(define-struct sammich (bread filling))

(define sam1 (make-sammich 'rye 'turkey))
(define sam2 (make-sammich 'white 'pbnj))
(define sam3 '(wheat pickles) )

sam1
sam2

(sammich? sam1)
(sammich? sam3)

(sammich-bread sam1)
(sammich-filling sam2)

(printf "sam1 is ~a on ~a\n"
  (sammich-filling sam1) (sammich-bread sam1))
(printf "sam2 is ~a on ~a\n"
  (sammich-filling sam2) (sammich-bread sam2))
(set! sam1 sam2)
(printf "sam1 is ~a on ~a\n"
  (sammich-filling sam1) (sammich-bread sam1))
```

Output

```
#<sammich>
#<sammich>

#t
#f

'rye
'pbnj

sam1 is turkey on rye
sam2 is pbnj on white

sam1 is pbnj on white
```

Example: Sammich Forgery Attempt

```
#lang racket ;; This is "sammich.rkt"
(provide awesome-sammich print-sammich)
(define-struct sammich (bread filling))
(define (awesome-sammich) (make-sammich 'rye 'turkey))
(define (print-sammich sam)
  (cond
    [(sammich? sam)
        (printf "~a on ~a\n" (sammich-filling sam)
    (sammich-bread sam))]
    [else (printf "I did not make that sammich!!\n")]))
```

```
#lang racket ;; This is "sammichtest.rkt"
(require "sammich.rkt")
(define s1 (awesome-sammich))
(print-sammich s1)
(define-struct sammich (bread filling))
(define sforge (make-sammich 'croissant 'pickles))
(print-sammich sforge)
```

```
turkey on rye
I did not make that sammich!!
```

Observations

- Because the `sammich` functions (`make-sammich`, `sammich-bread`, etc.) are not provided by the module, the client can't use them:
  - Can't take apart a sammich.
  - Can't build a new sammich.
  - ... not even if the client creates his/her own `sammich` structure!
- Racket recognizes the difference between the functions `make-sammich`, etc., defined **inside** the module, and functions with the same name defined **outside** the module.

A more secure queue implementation

In file squeue.rkt:

```
#lang racket  ;; "squeue.rkt"
(provide new-queue queue-empty?
  enqueue head dequeue)

(define-struct queue (lst))

(define (new-queue) (make-queue empty))
(define (queue-empty? q)
  (empty? (queue-lst q)))
(define (enqueue q item)
  (make-queue (cons item (queue-lst q))))
(define (head q)
  (last (queue-lst q)))
(define (dequeue q)
  (make-queue (drop-right (queue-lst q) 1)))
;;Note make-queue is not the same as new-queue
```

Attempted Queue Abuse

Testing Code

```
#lang racket  ;; "queue-ex3.rkt"
(require "squeue.rkt")
(define Q
  (enqueue
    (enqueue (new-queue) 'tomato)
                           'onion))
;;Postconditions imply n = 2 here

;; I hate tomatoes!!
Q
(set! Q (list (first Q)))

(queue-empty? Q)
(head Q)
(define Q2 (dequeue Q))
(head Q2)
(define Q3 (dequeue Q2))
;;Postconditions imply n = 0 here
(queue-empty? Q3)
```

Output

```
#<queue>
first: expected argument
of type <non-empty list>;
given #<queue>
```

Data Hiding

- Structures in Racket are (by default) *opaque*: we can't look inside them except for the functions we provide
- The advantage of using an opaque structure to hide the list is that code outside cannot create or operate on queues in any way other than through the functions that queue.rkt provides.
- If we later change the implementation in queue.rkt, it will not break other code.

Mutable ADTs

Mutable ADTs may be modified to change the data they hold.

- Our queue operations always returned a new queue, e.g.
  - enqueue : Queue Any -> Queue
    - Two parameters, an item e and a queue $Q = (q_1, q_2, ..., q_n)$
    - PRE: True
    - POST: Produces $Q' = (e, q_1, q_2, ..., q_n)$
- Sometimes it is useful if we actually modify the queue that is passed to the ADT operation. I.e., change Q in the queue ADT.
- Since its an *abstract* data type, this queue being modified is sometimes called the *state* associated with the queue.
- This modification is a kind of side effect, and must be carefully specified.
  - POST: Modifies Q so now $Q = (e, q_1, q_2, ..., q_n)$

Mutable Structs in Racket

```
(define-struct mytype (field-a field-b ...) #:mutable)
```

- A special form which defines the following functions:
  - Constructor: `(make-mytype val-for-a val-for-b ...)`
  - Accessors: `(mytype-field-a the-mytype)`, `(mytype-field-b the-mytype)`, ...
  - Type-checker: `(mytype? the-thing)`
  - **Mutators:** `(set-mytype-field-a! the-mytype new-a-value)`, `(set-mytype-field-b! the-mytype new-b-value)`, ...
    - Takes an existing struct, rebinds the field to the new value.

Example: The Mutable Sammich

Output:

```
sam1 is turkey on rye
sam1 is ham on rye
```

Example Code

```
#lang racket
(define-struct sammich (bread filling) #:mutable)
(define sam1 (make-sammich 'rye 'turkey))

(printf "sam1 is ~a on ~a\n"
  (sammich-filling sam1) (sammich-bread sam1))

(set-sammich-filling! sam1 'ham)

(printf "sam1 is ~a on ~a\n"
  (sammich-filling sam1) (sammich-bread sam1))
```

Example: The Mutable Queue ADT

In contracts and data definitions, we will use the exclamation point (!) when naming types that are mutable and when naming operations that peform mutation.

A Queue! is a (possibly empty) sequence $(q_1, q_2, ..., q_n)$.

Operations

- new-queue! : $\rightarrow$ Queue!
  - PRE: True
  - POST: Produces a new empty Queue!
- queue!-empty? : Queue! $\rightarrow$ Boolean
  - One parameter, a queue $Q = (q_1, q_2, ..., q_n)$
  - PRE: True
  - POST: Produces True if sequence is empty, False otherwise
- ...

Mutable Queue: More interesting operations

- enqueue! : Queue! Any $\rightarrow$ Void
  - Two parameters, an item e and a queue $Q = (q_1, q_2, ..., q_n)$
  - PRE: True
  - POST: **Modifies** Q so that now $Q = (e, q_1, q_2, ..., q_n)$
- head : Queue! $\rightarrow$ Any
  - One parameter, a queue $Q = (q_1, q_2, ..., q_n)$.
  - PRE: $n \geqslant 1$
  - POST: Produces value q_n.
- dequeue! : Queue! $\rightarrow$ Void
  - One parameter, a queue $Q = (q_1, q_2, ..., q_n)$
  - PRE: $n \geqslant 1$
  - POST: **Modifies** Q so that now $Q = (q_1, ..., q_{n-1})$.

A Mutable Queue Implementation in Racket

```racket
#lang racket  ;; "mutqueue.rkt"
(provide new-queue! queue!-empty?
  enqueue! head dequeue!)

(define-struct queue! (lst) #:mutable)

(define (new-queue!) (make-queue! empty))

(define (queue!-empty? q) (empty? (queue!-lst q)))

(define (enqueue! q item)
  (set-queue!-lst! q
    (cons item (queue!-lst q))))

(define (head q) (last (queue!-lst q)))

(define (dequeue! q)
  (set-queue!-lst! q
    (drop-right (queue!-lst q) 1)))
```

Mutable Queue Use

Testing Code

```
#lang racket  ;; "mutqueue-ex.rkt"
(require "mutqueue.rkt")
(define Q (new-queue!))
(queue!-empty? Q)
(enqueue! Q 'carrot)
(enqueue! Q 'tomato)
(enqueue! Q 'onion)
;;Postconditions imply n = 3 here
(queue!-empty? Q)
(head Q)
(dequeue! Q)
(head Q)
(dequeue! Q)
(head Q)
(dequeue! Q)
;;Postconditions imply n = 0 here
(queue!-empty? Q)
```

Output

```




#t
    
    
    
    
#f
'carrot
  
'tomato
  
'onion
  
  
#t
```

Hold on just a minute.

```
...
(define (enqueue! q item)
  (set-queue-lst! q
    (cons item (queue-lst q))))
...
```

- What is going on here? Are we any further ahead?
- We have two versions of the queue ADT, one which mutates its parameters, one which does not
- Underneath, their implementations are essentially the same
- We will see that there are advantages to each implementation.
- Some of these advantages are programming language specific.

Efficiency of our queue implementation

- One of the motivations for ADTs was that we can substitute a more efficient implementation for a less efficient one.
- In our implementation all the operations except `dequeue` and `head` require a small constant number of basic operations.
- `dequeue` relied on `drop-right`

```
(define (dequeue q)
  (drop-right q 1))
```

  This requires time roughly proportional to the length of the queue. Why?
- It would be nice to have all operations require a constant amount of time.

Back to the passport office

- Needed a way to keep track of
  - The order in which people came in – gave numbers to "clients"
  - The order in which we serve people – gave numbers to "servers"
- It took us a lot of legwork to get this operational!
- Need to keep track of people, make sure the first one in is the first one out
- First-In-First-Out ➡ FIFO
- This is exactly a queue's job!
- We shall bend it to our will...

The new ADT-based passport office

```
#lang racket  ;; "passport-mutqueue.rkt"
(require "mutqueue.rkt")
(define (passport-office)
  (define pq (new-queue!))
  (passport-office-helper pq))
(define (passport-office-helper pq)
  (printf "Enter a command (enter,serve,quit): ")
  (define c (read))   ;; read a command
  (cond
    [(eq? c 'enter) (define instd (read)) (enqueue! pq instd)
       (printf "inserted ~a\n" instd)
       (passport-office-helper pq)]
    [(eq? c 'serve)
       (cond
         [(queue!-empty? pq) (printf "Nobody to serve\n")
                             (passport-office-helper pq)]
         [else (printf "Now serving: ~a\n" (head pq))
               (dequeue! pq)
               (passport-office-helper pq)])]
    [(eq? c 'quit) (printf "Have a nice day!\n")]
    [else (printf "Invalid command!\n")
          (passport-office-helper pq)]))
(passport-office)
```

Another Example: The (Mutable) Stack ADT

Another important example of an ADT is a *stack*.
A Stack! is a (possibly empty) sequence $(s_1, s_2, ..., s_n)$ with a Last-In-First-Out (LIFO) semantics—we can only remove the most recently-inserted item from a stack.

Operations

- new-stack! : $\rightarrow$ Stack!
  - PRE: True
  - POST: Produces a new empty Stack!
- stack!-empty? : Stack! $\rightarrow$ Boolean
  - One parameter, a Stack! $S = (s_1, s_2, ..., s_n)$
  - PRE: True
  - POST: Produces True if sequence is empty, False otherwise
- ...

Stack: More interesting operations

- push! : Stack! Any $\rightarrow$ Void
  - Two parameters, an item e and a stack $S = (s_1, s_2, ..., s_n)$
  - PRE: True
  - POST: Modifies S so that $S = (e, s_1, s_2, ..., s_n)$
- top : Stack! $\rightarrow$ Any
  - One parameter, a stack $S = (s_1, s_2, ..., s_n)$.
  - PRE: $n \geqslant 1$
  - POST: Produces value s_1.
- pop! : Stack! $\rightarrow$ Void
  - One parameter, a stack $S = (s_1, s_2, ..., s_n)$
  - PRE: $n \geqslant 1$
  - POST: Modifies S so that $S = (s_2, ..., s_n)$.

The Stack ADT

- It is easy to imagine an implementation module:

```
(provide new-stack! stack!-empty? push! top pop!)
(define-struct stack! (lst) #:mutable)

... ; definitions of new-stack, stack-empty?, top

(define (push! the-stack item)
  (set-stack!-lst! the-stack
      (cons item (stack!-lst the-stack))))
(define (pop! the-stack)
  (set-stack!-lst! the-stack (rest (stack!-lst the-stack))))
```

- Note that `push!` and `pop!` are even simpler than `enqueue!` and `dequeue!` in our implementation of the `queue` ADT
- We could also define an immutable stack ADT by simply returning new stacks, instead of modifying the existing one.

The Sequence ADT

Lists are the fundamental data structure in Racket.

- We can view lists as a concrete implementation of a more general notion of an ADT Sequence.
- Later, we will see other ways to implement this ADT.

A *Sequence* $s_0, \ldots, s_{n-1}$ provides the following operations:

Basic Operations – immutable sequence

- new-sequence: $\rightarrow$ Sequence
  - Parameters: None
  - PRE: True
  - POST: produces an empty sequence
- sequence-length: Sequence $\rightarrow$ Nat
  - Parameters: a sequence $s = (s_0, \ldots, s_{n-1})$
  - PRE: True
  - POST: Produces n

ADT Sequence (continued)

- item-at: Sequence Nat $\rightarrow$ Any
  - Parameters: a sequence $s = (s_0, \ldots, s_{n-1})$ and a natural number i
  - PRE: $i < n$
  - POST: Produces s_i
- insert-at: Sequence Nat Any $\rightarrow$ Sequence
  - Parameters: a sequence $s = (s_0, \ldots, s_{n-1})$, a natural number i, and an item e.
  - PRE: $i \leqslant n$
  - POST: Produces a new sequence $s' = (s_0, \ldots, s_{i-1}, e, s_i, \ldots, s_{n-1})$
- remove-at: Sequence Nat $\rightarrow$ Sequence
  - Parameters: a sequence $s = (s_0, \ldots, s_{n-1})$ and a natural number i.
  - PRE: $i < n$
  - POST: Produces a new sequence $s' = (s_0, \ldots, s_{i-1}, s_{i+1}, \ldots, s_{n-1})$

Think about how a *Mutable Sequence ADT* might be defined.

Summary

Separations of concerns

- Keep implementation separate from interface
- When applied to data: Abstract Data Types (ADTs)
  - Tell client how to use the data, not what it looks like
  - Keep implementation details secure; prevent tampering and forgery
  - Queues, Stacks, Sequences
- Mutable vs. immutable data structures