CS60 Cheat Sheet

Aaron Zhang

Basics & Recursion:

Recursion Template:

```
define (f input)
  (cond
    [(base-condition? input) base-result]   ← base case
    [else (combine current-part (f (smaller input)))])) ← recursive case
```

Recursion Example:

```
(define (triangle n)
  (if (= n 0)
      0                                  ← for lists: '(), for strings: ""
      (+ n (triangle (- n 1)))))
```

↳ for lists, for strings: string-append

↓

cons: creates new list (used most of the time)
append: joins two lists together (used if building backwards)

Use it or lose it example:

```
(define (group-sum L target)
  (if (empty? L)                         } base case: check if L is empty and if the
      (= target 0)                       } target = 0 (meaning true), otherwise false
      (or (group-sum (rest L) (- target (first L)))   } either use or lose,
          (group-sum (rest L) target))))              } covering all possibilities w recursion
```

HOFs:

HOF Examples:

map: takes in function and list, applies function to every element of list

```
(def (last-digit x)
  (modulo x 10))
(map (last-digit '(1 22 93)) ← applies last-digit to every element
→ '(1, 2, 3)
```

filter: takes function that returns true/false and a list

```
(define (greater-than-3 x)
  (> x 3))
(filter greater-than-3 '(1, 2, 3, 4, 5)) ← applies greater-than-3 to every element, returns all inputs that evaluate to true
→ '(4, 5)
```

fold l/r: collapse list into one value

```
(define (my-add a b)
  (+ a b))
(foldr my-add 0 '(1, 2, 3, 4))   ← base value (being added to)
→ 10
```

r vs. l: builds from left (l) or right (r)

Use cases: foldr: rebuilding list (same order) (foldr cons '()), foldl: ↖

↳ adds to the front

Lambda:

Template: (lambda (x) (do something w x))

Example: (lambda (x) (+ 1 x)) ← add to input

Loop Counting:

Example:

```
for (int i=0; i<n; i++){
    for (int j=i; j<n; j++){      ← Nested for loop, for each i, j+1 until j<n
        count++; }}
```

Counting:

i = 0	j = 0 to N-1	steps/i: N	Summation:
i = 1	j = 1 to N-1	steps/i: N-1	N + N-1 + N-2 ... + 2 + 1
i = 2	j = 2 to N-1	steps/i: N-2	= $\frac{N(N-1)}{2}$
...		...	
i = N-1	j = N-2 to N-1	steps/i: 2	
i = N-2	j = N-1 to N-1	steps/i: 1	= $O(N^2)$

Function Tracing:

```
(define (reverse L)
  (if (empty? L)
      L
      (append (reverse (rest L))
              (list (first L)))))
```

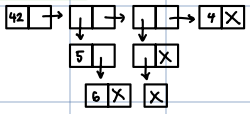
Tracing:

```
>(reverse '(2,3))   } inputs
>(reverse '(3))     }
>(reverse '())

<'()                } outputs
<'((3))             }
<'((3 (2 3))
'((3 (2 3))   ← final output
```

Lists:

Linked Lists



Rules:

→ = next element

↓ = rest of list

⊡ = sublist

= '(42 (5 6) (( )) 4)   ← output

Test Cases:

Test Cases Example:

```
public class Iterative {
  (returns true if even number present in array)
```

Test Case:

```
@Test                                  ← JUnit test method
public void test1() {                  ← Define test method
  int[] testValues = {1, 3, 7, 4};     ← create array of test values
  assertTrue(Iterative.f(testValues)); ← check if true by passing array to method
}
```

↑ for int[], this depends on the type of array

Other array types:

```
char[]
boolean[]
String[]
```

Big O:

Recurrence relationship:

Example:

For function O(1):

T(0) = 1 ← constant time

T(N) = 1 + T(N-1) ← constant time + rest

= O(N)

For function O(n):

T(0) = 1 ← Base case, prob not called

T(N) = N + T(N-1) ← Linear time + rest

= $O(N^2)$

Unrolling:

T(N-1) = N-1 + T(N-2)
T(N-2) = N-2 + T(N-3) } Cheat Sheet
T(N-3) = N-3 + T(N-4)

Result:

T(N) = N + T(N-1) ↘

T(N) = N + N-1 + T(N-2) ↘

T(N) = N + N-1 + N-2 + T(N-3)

T(N) = N + N-1 + N-2 + N-3 ... + 3 + 2 + 1

= $\frac{N(N+1)}{2}$ = $O(N^2)$