QUICK CONTENT REVIEW

- Intro to Scheme

( ) → evaluate

#f/#t → only false value is '#f', everything else is #t

primitives → you cannot evaluate them by putting parenthesis around them
↳ ie. (4) will cause error, equivalent to trying to write '4()' in python

compound expressions → call expressions & special forms
→ encapsulate multiple sub-expressions in parentheses which evaluates to a single value
→ call expressions
→ begin w/ function name, followed by arguments
(operator/function_name operand1/arg1 operand2/arg2 ...)

eg. (+ 3 2)
(go 3) → if function go(x) is already defined

→ order of evaluation
1) evaluate operator/function_name
2) evaluate operands/arguments
3) Apply function/operator to arguments/operands

eg. (run 2 3)
second, evaluate these arguments (just evaluate to themselves since they are primitives)
first, evaluate this name to figure out which function we are looking at and to determine how many operands we need to look at
third, evaluate the function named run with parameters = 2 & 3

→ special Forms
→ begin with some keyword you look at and then evaluate subexpressions in some way based on that specific keyword

eg. 'and' keyword evaluates the following expressions left to right, stopping after the first expression that evaluates to a false value

conditionals → (if condition return else-return)

→ no elif so instead of tons of nested if's, use cond keyword
(cond (cond1 return)
(cond2 return)
⋮)

boolean checks → = only works for integers/numbers
→ eq? is analogous to is in python
→ equal? is analogous to ==

(*) main special expressions
(define (function-name arg1 arg2 ...) function-body)
↳ also returns function-name & have to call again w/ operators

(define variable-name value)

↳ this would return variable-name and evaluating this variable-name again would then return the value

(lambda (arg1 arg2 arg3 . . .) body)

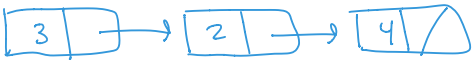
↳ creates a new lambda with args as the parameters and body as the body

- Pairs & Lists

lists → structure is similar to linked lists ("all lists in scheme are linked lists")

→ nested list construction

(cons 3 (cons 2 (cons 4 nil)))



→ way to create a new scheme list
'cons' is the keyword
(cons first_value rest_of_list)

→ simple construction w/ keyword list

(list 3 2 4)

→ car gets current value, cdr gets next value

```
>> (define x (list 3 2 4))
   x
>> (car x)
   3
>> (car (cdr x))
   2
```

PRACTICE PROBLEMS

Warm up : Q4

These short questions are meant to help refresh your memory of topics covered in lecture and lab this week before tackling more challenging problems.

Describe the difference between the following two Scheme expressions. Hint: which defines a new procedure?

Expression A:

```
(define x (+ 1 2 3))
```

Next, we evaluate (define x 6)

this is equivalent to x = 6 in python

we first evaluate this: 1+2+3 = 6

Expression B:

```
(define (x) (+ 1 2 3))
```

Next, we evaluate (define (x) 6)

difference from expression A is there extra parenthesis (special form ↳ defining function)

this is equivalent to the following in python:

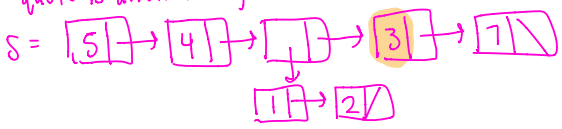
```
def x():
    return 6
```

Expression B defines a new procedure
↳ the function x()

Write an expression that selects the value 3 from the list below.

```
(define s '(5 4 (1 2) 3 7))
```

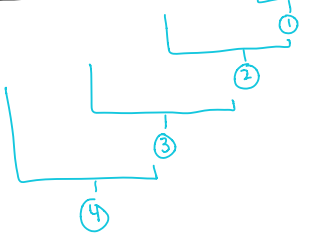
↑ quote is another way to construct a list in scheme

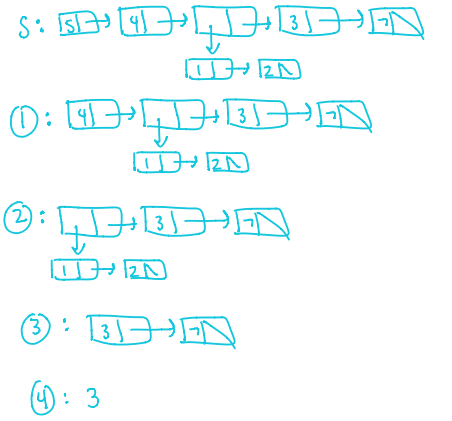


Remember 'car' gets the first value & 'cdr' gets the rest of the list

↳ to get 3:

```
(car (cdr (cdr (cdr s))))
```





quote
cons

(car lst) (cdr lst)

.first .rest

Link(value, rest)

(cons value rest)

car cdr

Fibonacci : Q2

Fibonacci numbers: 0, 1, 1, 2, 3, 5, 8, 13, ...
(0 1 2 3)

Tutorial: Write a function that returns the n^{th} Fibonacci number.

4:25

```
(define (fib n)
```

```
)
scm> (fib 0)
0
scm> (fib 1)
1
scm> (fib 10)
55
```

First lets write fibonacci in python together: (recursively)

```
def fib(n):
    if n <= 1:
        return n
    else:
        return fib(n-1) + fib(n-2)
```

Now, convert this into scheme!

```
(if cond
    → if true
    → if false )
```

```
(define (fib n)
    (if (<= n 1)
        n
        (+ (fib (- n 1)) (fib (- n 2)))
    )
)
```

SOLUTION:

```
(define (fib n)
    (if (<= n 1)
        n
        (+ (fib (- n 1)) (fib (- n 2)))
    )
)
```

Scheme ← Python

```
def fib(n):
    if n <= 1:
        return n
    else:
        return fib(n-1) + fib(n-2)
```

List Duplicator : Q5

Tutorial: Write a Scheme function that, when given a list (input), such as `(1 2 3 4)`, duplicates every element in the list (output) (i.e. `(1 1 2 2 3 3 4 4)`).

```
(define (duplicate lst)
)
```

4:37

Hints/Tips to get started...
- remember how to create a new list in scheme
  - → similar to LL
  - (cons val lst)
- how do we access the first & rest vals of a scheme list?
  - → car & cdr

SOLUTION:

```
(define (duplicate lst)
  (if (null? lst)          → base case
      lst                  → if lst is null, return lst cuz now we're done w/ our updates
      (cons (car lst) (cons (car lst) (duplicate (cdr lst))))
  )
)
```

- (cons (car lst) ...) → return a new list; 'cons' makes a new list
- (car lst) → the first element of new list should be first element of list so that it is duplicated
- (car lst) → we still want to keep the 'original' first element
- (cons ... (duplicate (cdr lst))) → 2nd argument to 'cons' just like linked list should be another list (recursion)
- (duplicate (cdr lst)) → recursive call on rest of lst

only do if have extra time at end!

List Insert: Q6

Q6: (Tutorial) List Insert

input → given an element, a list, and an index,
task → inserts the element into the list at that index.

Write a Scheme function that, when given an element, a list, and an index, inserts the element into the list at that index. You can assume that the index is in bounds for the list.

```
1  (define (insert element lst index)
2    'YOUR-CODE-HERE
3  )
4
```

4:52

SOLUTION:

```
(define (insert element lst index)
  (if (= index 0)
      (cons element lst)
      (cons (car lst) (insert element (cdr lst) (- index 1)))
  )
)
```

if we have reached the insertion index, create a new lst, that adds this element to the front of the "remaining" list

(cons) → make a new list

(car lst) → first element of new lst is same as passed in lst

(insert element (cdr lst) (- index 1)) → recursive call
↳ element = element
lst = car lst
index = index - 1