Scheme

- Call expressions in Scheme work exactly like they do in Python
  1. Evaluate operator
  2. Evaluate operands from left to right.
  3. Apply the value of the operator to the evaluated operands.
- (cdr (cons 1 (cons 2 nil))) ⟶ (2)
- (car (cdr '(127 . ((131 . (137)))))) ⟶ (car (cdr '(127 (131 137)))) ⟶ (131 137)
- (define (composed f g) (lambda (x) (f (g x))))
- ((+ 1 2)) ⟶ Error (int is not callable)
- (= 0 #t) ⟶ Error (#t is not a number)
- (if 0 'cond 'else) ⟶ cond
- (if (or #t (/ 1 0)) 1 (/ 1 0)) ⟶ 1
- =, eq?, equal?
  - = can only be used for comparing numbers.
  - eq? behaves like == in Python for comparing two non-pairs. Otherwise, eq? behaves like is in Python.
  - equal? compares pairs by determining if their cars and cdrs are equal (i.e. have the same contents). Otherwise, equal? behaves like eq?.

Interpreters, Tail Recursion

- REPL: Read-Eval-Print Loop
  - Read: Calls lexer, which breaks input into tokens. Calls parser, which organizes the tokens into data structures.
  - Eval: Mutual recursion between eval and apply. eval evaluates expressions (through lookups or by recursively calling eval and apply). apply applies an evaluated operator to its evaluated operands.
  - Print: Display the result of evaluating user input.
  - Loop: Repeat if there are more expressions to interpret.
- # of times scheme_apply: Use arrows for each function call (operators, exclusing special forms)
- # of times scheme_eval: Use underline for each expression (look for parentheses) and each value (operators + operands, but excluding special forms again)
- Special forms (not scheme_evaled or scheme_applied)
  - define
    - +1 scheme_eval for the entire expr
    - Variable: only evaluate expr you are binding to variable name, e.g. (define a 3) have another scheme_eval for 3
    - Function: no additional scheme_eval

    - For functions, only evaluate function body when function is called
  - define-macro, lambda, mu: no evaluation until called (only 1 scheme_eval for entire expr)
  - if, cond, and, or: short-circuiting
  - let: evaluate all exprs in bindings, then evaluate exprs in the body
  - begin: evaluate all operands
  - quote, quasiquote, unquote, unquote-splicing: no evaluation
  - cons-stream
  - delay, set!

- A tail call occurs when a func calls another func as its last action of the current frame. In this case, the frame is no longer needed, and we can remove it from memory, i.e. we can reuse the current frame instead of making a new frame.

- Tail-Recursion $\longrightarrow$ Pass-Down Recursion $\longrightarrow$ helper function + an argument that represents the "result so far"

Macros, Streams

- (define-macro (f x) (car x))
  - (f (+ 2 3)) $\longrightarrow$ #[+]
  - (f (x y z)) $\longrightarrow$ Error
  - (define x 2000) (f (x y z)) $\longrightarrow$ 2000
  - (f (list 2 3 4)) $\longrightarrow$ #[list]
  - (f (quote (2 3 4))) $\longrightarrow$ SchemeError

- Evaluating calls to macro procedures are:
  1. Evaluate operator
  2. Apply operator to unevaluated operands
  3. Evaluate the expression returned by the macro in the frame it was called in.
  4. For quasiquote, entire expr +1, scheme_eval any unquote and ignore others.

- Streams have two important properties: Lazy evaluation, Memoization

- cons-stream, car, cdr-stream (computes and returns the rest of stream), nil (empty-stream), cdr (gets rest of a stream wrapped in a promise)
  - (define (naturals n) (cons-stream n (naturals (+ n 1)))) $\longrightarrow$ naturals
    (define nat (naturals 0)) $\longrightarrow$ nat
    (car nat) $\longrightarrow$ 0; (cdr nat) $\longrightarrow$ #[promise (not forced)]
    (car (cdr-stream nat)) $\longrightarrow$ 1
    (car (cdr-stream (cdr-stream nat))) $\longrightarrow$ 2
    (cdr-stream nat) $\longrightarrow$ (1 . #[promise (forced)])
  - (define (compute-rest n) (print 'evaluating!) (cons-stream n nil)) $\longrightarrow$ compute-rest
    (define s (cons-stream 0 (compute-rest 1))) $\longrightarrow$ s
    (car (cdr-stream s)) $\longrightarrow$ evaluating! \n 1
    (car (cdr-stream s)) $\longrightarrow$ 1

- (define factorials (cons-stream 1 (combine-with * (naturals 1) factorials)))

- Macro that strips out every other argument:
  (define (prune lst) (if (or (null? lst) (null? (cdr lst))) lst (cons (car lst) (prune (cdr (cdr lst))))))
  (define-macro (prune-expr expr) (cons (car expr) (prune (cdr expr))))
  scm> (prune-expr (prune-expr (+ 10 100) 'garbage)) ⟶ 10
- (define (cadr lst) (car (cdr lst)))
  (define-macro (let-macro bindings body) (cons ` (lambda ,(map car bindings) ,body) (map cadr bindings)))

SQL

- Constraint: WHERE xxx LIKE '%[word]%'
- Order, Limit: ORDER BY [cols] (DESC) LIMIT [lim];
- More on Final Guide right column of page 1
- INSERT INTO table SELECT ... OR INSERT INTO table VALUES etc. (Final Guide)
- (String) concatenation operator: ||
- SELECT *, COUNT(*) represents choosing/counting all rows
- Aggregate functions: MAX, MIN, COUNT, SUM, AVG, etc.
- Remember to end each statement with a ;
- Declarative programming language (Python, Scheme are imperative)

Extra Sanity Checks

- In Python, func map and filter returns an iterator
- Iterable includes iterator, list, string, generator (anything that could be called iter on)
- If a = iter(iterable), then a is iter(a)
- For list mutation (with recursion especially), care about the pointer, i.e. can't reassign passed in lst directly, has to change lst.first and lst.rest
- When removing objects inside a for loop, i.e. deleting a branch from a tree, use "for b in list(t.branches)" or "for b in t.branches.copy()" or "for b in t.branches[:]"
- str('a') ⟶ 'a'
- Previously... HOF always write another layer outside (for reference issues)

```
def repeated(f):
    g = lambda x: x
    while True:
        yield g
        g = (lambda g: lambda x: f(g(x)))(g)
```

- Scheme

  - built-in functions: =, quotient, modulo, even?, odd?, etc. (expt 3 5) = 3^5
  - (null? lst) checks if lst is nil
  - Unlike in Python, the only primitive in Scheme that is a false value is #f and its equivalents, false and False. This means that 0 is not false. (false/'false $\longrightarrow$ #f)
  - #t, #f can't be operated by $=, >, <$ (with nums or themselves), i.e. $=, >, <$ can only operate on true numbers
  - Macros do not have their arguments evaluated, whereas normal Scheme procedures do. Macros have their return value re-evaluated a second time, whereas normal Scheme procedures do not. Normal Scheme procedures take in values and return values. Macros can be seen as taking in code and returning code.
  - Scheme $\sim$ Python: (cons elem lst) $\sim$ [elem] + lst; (list elem1 elem2) $\sim$ [elem1, elem2]

- SQL
  - Remember to end each statement with a ;
  - Not equal: != or <>

- Expected topics (if have absolutely no idea, check if one of the topics haven't been tested on):
  1. WWPD
  2. Env. Diagram
  3. Python Lists, Mutation
  4. Tree Recursion
  5. Generators
  6. Scheme
  7. SQL