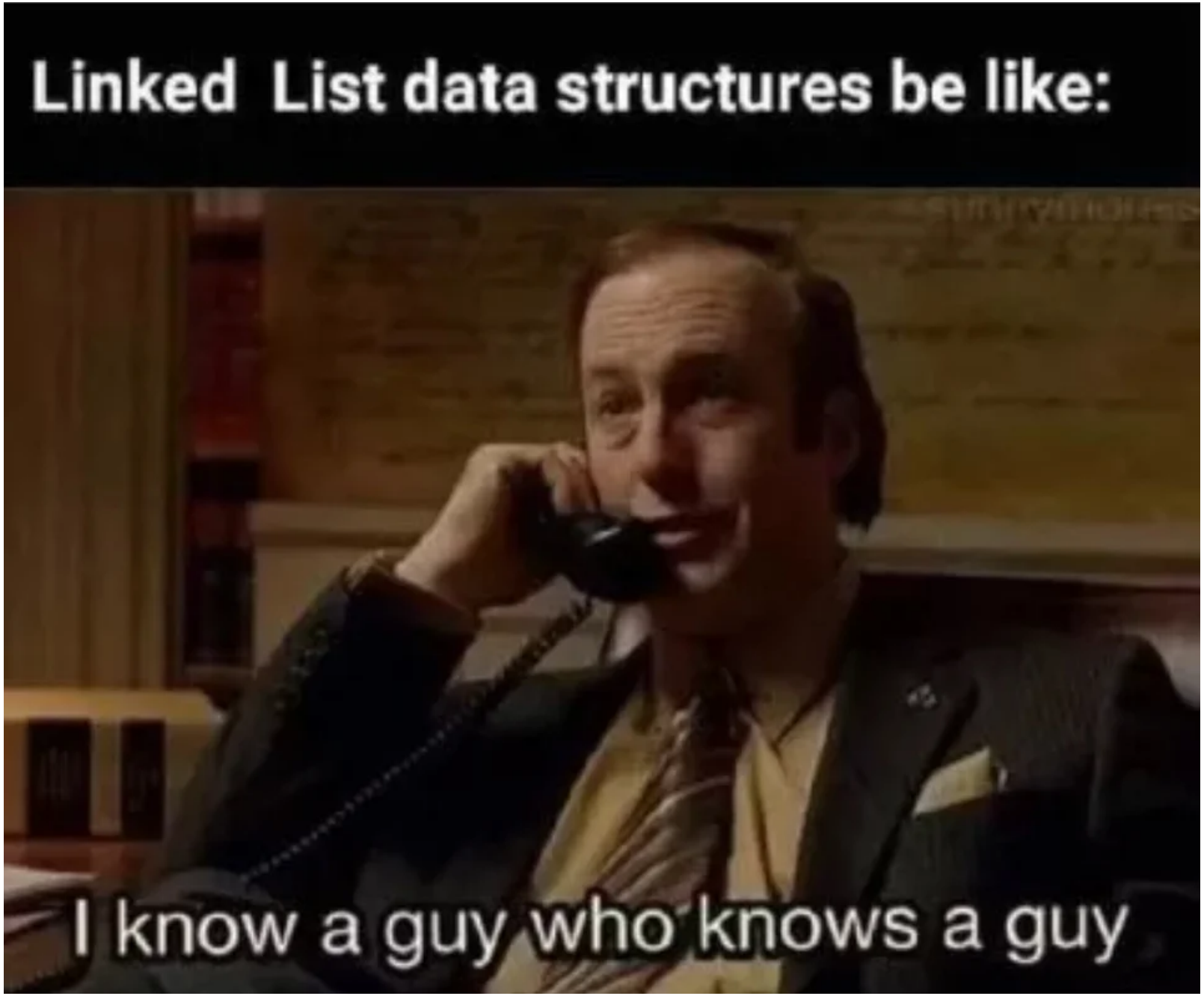
Linked List data structures be like:
I know a guy who knows a guy

Q3: (Tutorial) Reverse

Write a tail-recursive function `reverse` that takes in a Scheme list a returns a reversed copy.

Hint: use a helper function!

```
(define (reverse lst)
'YOUR-CODE-HERE

)
```

```
scm> (reverse '(1 2 3))
(3 2 1)
scm> (reverse '(0 9 1 2))
(2 1 9 0)
```

Q5: (Tutorial) If Macro Python

In Homework 1 you implemented an "If function" that functioned differently from an if statement. It was different because it did not short circuit in evaluating its operands! In this problem, we will write a "macro" in Python called `if_macro` that takes in three arguments:

1. `condition`: a string that will evaluate to a truth-y or false-y value
2. `true_result`: a string that will be evaluated and returned if `condition` is truth-y
3. `false_result`: a string that will be evaluated and returned if `condition` is false-y. and returns a **string** that, when evaluated, will return the result of this if function.

`if_macro` should **only** evaluate `true_result` if `condition` is truth-y, and will only evaluate `false_result` if `condition` is false-y.

Hint: You can write a one-line if statement with the following syntax: `value_when_true` if `condition` else `value_when_false`

Hint: Using f-Strings might make this problem easier too!

```
def if_macro(condition, true_result, false_result):
    """
    >>> eval(if_macro("True", "3", "4"))
    3
    >>> eval(if_macro("0", "'if true'", "'if false'"))
    'if false'
    >>> eval(if_macro("1", "print('true')", "print('false')"))
    true
    >>> eval(if_macro("print('condition')", "print('true_result')", "print('false_result')"))
    condition
    false_result
    """
    "*** YOUR CODE HERE ***"
```

Q5: (Tutorial) If Macro Python

In Homework 1 you implemented an "If function" that functioned differently from an if statement. It was different because it did not short circuit in evaluating its operands! In this problem, we will write a "macro" in Python called `if_macro` that takes in three arguments:

1. `condition`: a string that will evaluate to a truth-y or false-y value
2. `true_result`: a string that will be evaluated and returned if `condition` is truth-y
3. `false_result`: a string that will be evaluated and returned if `condition` is false-y. and returns a **string** that, when evaluated, will return the result of this if function.

`if_macro` should **only** evaluate `true_result` if `condition` is truth-y, and will only evaluate `false_result` if `condition` is false-y.

Hint: You can write a one-line if statement with the following syntax:
`val_when_true` if `condition` else `val_when_false`

Hint: Using f-Strings might make this problem easier too!

```
def if_macro(condition, true_result, false_result):
    """
    >>> eval(if_macro("True", "3", "4"))
    3
    >>> eval(if_macro("0", "'if true'", "'if false'"))
    'if false'
    >>> eval(if_macro("1", "print('true')", "print('false')"))
    true
    >>> eval(if_macro("print('condition')", "print('true_result')", "print('false_result')"))
    condition
    false_result
    """
    "*** YOUR CODE HERE ***"
```

Q7: (Tutorial) If Macro Scheme

Now let's try to write out the if macro in scheme! There's quite a few similarities between Python and Scheme, but we do have to make a few adjustments when converting our code over to Scheme. We'll start out by writing a scheme function using the `define` form we use for normal functions.

In the Python If macro, we returned a **string** that, when evaluated, would execute an if statement with the correct parameters. In Scheme, we won't return a string, but rather we'll return something else that represents an unevaluated expression. What type wil we return for scheme? Here, what "type" refers to what data type -- i.e. function, list, integer, string, etc..

In the Python If macro, our parameters were all **strings** representing the condition, return value if true, and return value if false. What type will each of our parameters be in Scheme? Give an example of an acceptable parameter for the condition.

Q7: (Tutorial) If Macro Scheme

Let's start writing this out!

Write a function `if-function` using the `define` form (not the `define-macro` form), which will take in the following parameters:

1. `condition` : a quoted expression which will evaluate to the condition in our if expression
2. `if-true` : a quoted expression which will evaluate to the value we want to return if true
3. `if-false` : a quoted expression which will evaluate to the value we want to return if false

and returns a Scheme list representing the expression that, when evaluated, will evaluate to the result of our if expression.

```
(define (if-function condition if-true if-false)
'YOUR-CODE-HERE









)
```

```
scm> (if-function '(= 0 0) '2 '3)
(if (= 0 0) 2 3)
scm> (eval (if-function '(= 0 0) '2 '3))
2
scm> (if-function '(= 1 0) '(print 3) '(print 5))
(if (= 1 0) (print 3) (print 5))
scm> (eval (if-function '(= 1 0) '(print 3) '(print 5)))
5
```

Q7: (Tutorial) If Macro Scheme

That felt a bit overly complicated just to create a function that emulates the if special form. We had to quote parameters, and we had to do an extra call to `eval` at the end to actually get our answer. To make things easier, we can use the `define-macro` special form to simplify this process.

As a reminder, the `define-macro` form changes the order of evaluation to be:

1. Evaluate operator
2. Apply operator to unevaluated operands
3. Evaluate the expression returned by the macro in the frame it was called in.

As a comparison, here are differences between `define-macro` and `define`:

1. The operands are **not** evaluated immediately. Instead, we can think of the operands as being quoted, similar to what we did in the previous part.
2. The return value gets `eval`ed at the very end, after the entire return expression is constructed. This means that we no longer have to call `eval` on the return value of our `if-function`.

Q7: (Tutorial) If Macro Scheme

Now, use the `define-macro` special form to define a macro, `if-macro`, that will have functionality shown by the following doctests (notice how the operands are no longer quoted, and that the final `eval` is gone):

```
(define-macro (if-macro condition if-true if-false)
'YOUR-CODE-HERE









)

scm> (if-macro (= 0 0) 2 3)
2
scm> (if-macro (= 1 0) (print 3) (print 5))
5
```