Q3: Reverse

Q3: (Tutorial) Reverse

very last thing evaluated is the recursive call

Write a tail-recursive function reverse that takes in a Scheme list a returns a reversed copy. Hint: use a helper function!

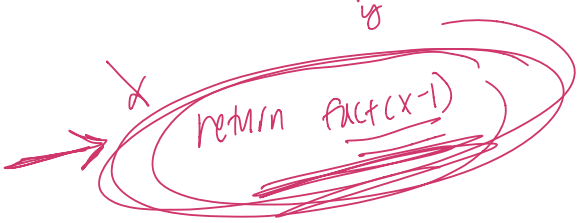
very common way to make any function tail recursive!

input output

```
1  (define (reverse lst)
2    'YOUR-CODE-HERE
3  )
4
```

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

```
def fact(x):
    return x * fact(x-1)
```

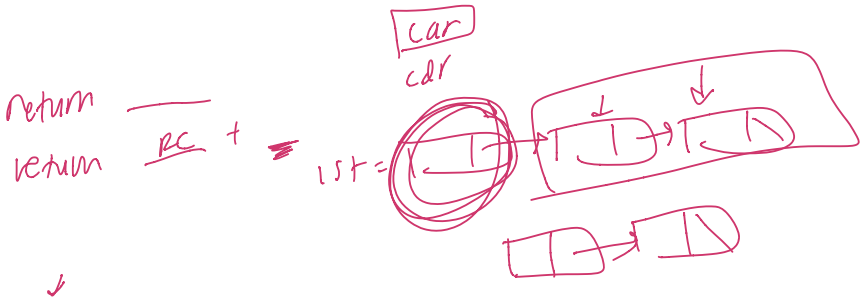


Link in chat!
↳ please write your solution/ideas there!!

work time: 5 min

4:18

SOLUTION:




```
(define (reverse lst)
  (define (reverse-tail sofar rest)
    (if (null? rest)
        sofar
        (reverse-tail (cons (car rest) sofar) (cdr rest))
    )
  )
  (reverse-tail nil lst)
)
```

→ create helper function

→ if there are no more elements (we've added them all to reverse copy, sofar), then just returns sofar

→ begin constructing reversed list (building the list up backward by always putting the next element in front of all the previous ones

tail recursive because the last thing to be evaluated in reverse-tail function is the recursive call

ie.) lst = 1 → 2 → 3

1st call to reverse-tail: sofar = 1

2nd call to reverse-tail: sofar = 2 → 1

Python Macros

Imagine if we wanted to write a function in python that would create and evaluate the following line of code:

```
[<expr> for i in range(5)]
```

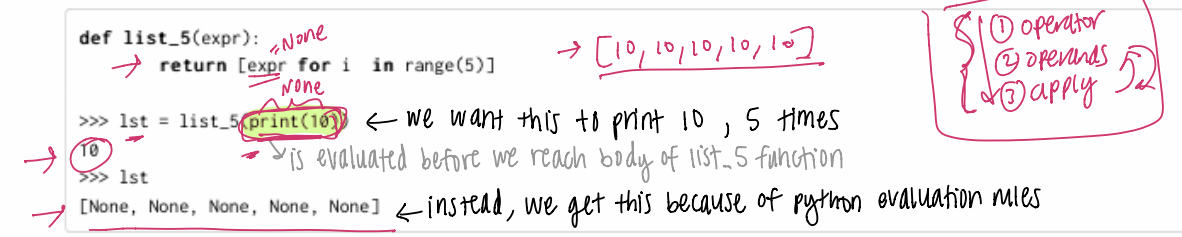
Where we could pass in any arbitrary expression `<expr>`, that would then be evaluated 5 times and have the results listed. This is the kind of problem that macros were made for!

A macro is a procedure that operates on unevaluated code. In the example above, `<expr>` is not necessarily an object, but could be any piece of code, such as `print("Hello")`, `[j ** 2 for j in range(i)]`, or `i >= 5`. For these expressions it is important to our problem statement that they not be evaluated until *after* they have been inserted into our list comprehension, either because they have side effects that will be apparent in the global frame, or because they depend on `i` in some way (there can be other reasons too!). So, we need a procedure that, instead of manipulating objects, manipulates code itself. This is where macros come in.

Unfortunately for us, Python does not have the ability to make true macros. Let's see what happens if we try to solve this problem with a traditional function.

```
def list_5(expr):
    return [expr for i in range(5)]

>>> lst = list_5(print(10))
10
>>> lst
[None, None, None, None, None]
```

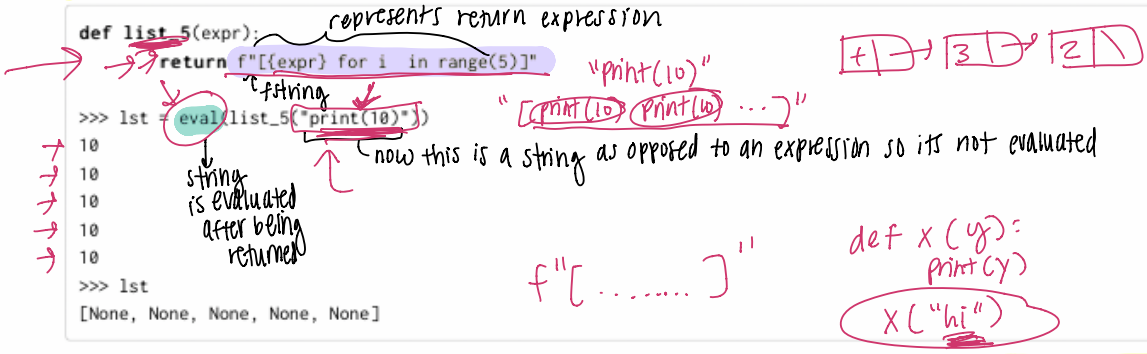


This isn't quite what we want. Because of Python evaluation rules, instead of evaluating a list of 5 `print(10)` statements, our `expr` was evaluated before the function was ever called, meaning 10 was only printed once.

The issue here is order of evaluation---we don't want our expression parameter to be evaluated until *after* it is inserted into our list comprehension. Even though we don't have the ability to make real macros when writing Python, we can write a function in the spirit of macros by returning a **string** representing the return expression, and **evaluating** this string after it has been returned.

```
def list_5(expr):
    return f"[{expr} for i in range(5)]"

>>> lst = eval(list_5("print(10)"))
10
10
10
10
10
>>> lst
[None, None, None, None, None]
```



Now we see the number 10 printed 5 times as a side effect of our function, just like we want! We circumvented Python's evaluate-operands-before-evaluating-function body rule by passing in our desired expression as a string. Then, after we constructed our list comprehension in string form and we've returned it, we used the `eval` function to force evaluation of the output after the last step in the execution of this function. We were able to write code that took code as a parameter and restructured it in the form of new code!

Although this is a cool hack we can do with Python, its usage is unfortunately rather limited. The `eval` function will throw a `SyntaxError` if it is passed any "compound" blocks such as `if: ...`, `while: ...`, `for: ...`, etc. (this does not include list comprehensions, or one-line if-statements such as `1 if a > b else 0`.)

Q5: If Macro Python

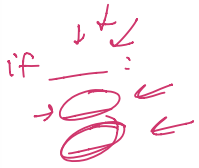
Q5: (Tutorial) If Macro Python

→ we want to implement 'if' without short circuiting

In Homework 1 (https://cs61a.org/hw/hw01/#q5) 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.



① operator
② operands
③ apply

> **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!

f "hi {condition}"

```
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 ***"
```

(#) Link in chat!
↳ please write your solution/ideas there!!

work time: 5 min

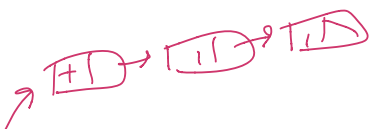
(#) working in python here!

4:43

SOLUTION:

```
def if_macro(condition, true_result, false_result):
    return f"{true_result} if {condition} else {false_result}"
```

" "



Q7: If Macro Scheme

(+ (+ 1 1) 2) ⇒ 

@Walk Through Together.@

Q7: (Tutorial) If Macro Scheme

in scheme, instead of a string we use a scheme list of pairs
- quoting (means don't evaluate) → ''
- quasiquoting (allows for unquoting within expression) → f-string

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.

Discuss the following questions with your tutorial group regarding how we can write out a Scheme if-function similar to our Python one:

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..

Your Answer:

① Scheme list

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.

'(= 1 1)

Your Answer:

② Quoted expression

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-func cond true false)
)

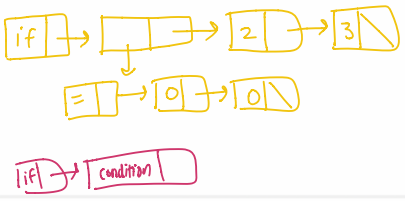
Here are some doctests to show this:

```
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
```

condition, if-false, if-true

will actually evaluate the 'if' form

let's draw it out!



③

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

'(if , condition , if-true , if-false)

↑ evaluation will replace the variable name condition with the actual string passed in (condition =)

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` :

takes care of quoting for us!

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`.

we don't need to explicitly call eval

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):

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

(4)

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

`'(if ,condition ,if-true ,if-false)`

notice how the body of the function didn't change! what changed is the way in which we call it & use it (no quotes & no eval explicitly)