

TAIL RECURSION, INTERPRETERS

CS61A SMALL GROUP TUTORING

July 27-29, 2022

1 Tail Recursion

Tail Recursion Overview

Often, when we write recursive functions, they can take up a lot of space by opening a bunch of frames. Think about `factorial(6)`. In order to solve it, we will have to open 6 frames. Now what if we tried `factorial(1000000)`? To avoid opening 1,000,000 frames, we can use a method called **tail recursion**. In tail call optimized languages, like Scheme (but not Python), tail recursive functions only use a **constant** amount of space. The key to defining a tail recursive function is to make sure no further calculations are done after the recursive call, so that none of the values in the current frame have to be saved. If we don't have to save any values in the current frame, we can close it as we make the next recursive call, ensuring that we only have one frame open.

In order to identify whether a function is tail recursive, first find the recursive call in your function. Then, check whether you return the exact result of your recursive call, or if you do work on the result. If you simply return the result of your recursive call, then your function is tail recursive! However, if you do additional work to the result of your recursive call, then it is not tail recursive. Additional work could be adding one to the result of your recursive call and returning the new value, or appending it to a list and returning the resulting list.

The general way we convert a recursive function to a tail recursive one is to move the calculation outside the recursive call into one of the recursive call arguments to accumulate the results. However, this is not always possible if our function doesn't have an argument that accumulates the results, so we may have to create a helper function with an accumulating argument and have the helper be a tail recursive function.

-
- Page 2

3. Consider the following function:

```
(define (count-instance lst x)
  (cond ((null? lst) 0)
        ((equal? (car lst) x) (+ 1 (count-instance
                                   (cdr lst) x)))
        (else (count-instance (cdr lst) x))))
```

What is the purpose of `count-instance`? Is it tail recursive? Why or why not?

Optional: draw out the environment diagram of this `count-instance` with `lst = (1 2 1)` and `x = 1`.

4. Rewrite `count-instance` to be tail recursive. (Hint: helper functions are often useful in implementing tail recursion.)

```
(define (count-tail lst x)
```

)

5. Implement `filter-lst`, which takes in a one-argument function `f` and a list `lst`, and returns a new list containing only the elements in `lst` for which `f` returns true. Your function must be tail recursive.

You may wish to use the built-in `append` function, which takes in two lists and returns a new list containing the elements of the first list followed by the elements of the second.

```
;Doctests
```

```
scm> (filter-lst (lambda (x) (> x 2)) '(1 2 3 4 5))  
(3 4 5)
```

```
(define (filter-lst f lst)
```

```
)
```

2 Interpreters

Interpreters Overview

An **interpreter** is essentially a program that understands and processes other programs.

The interpreter design we will be covering in 61A is the **Read-Eval-Print Loop**, which consists of the following steps:

1. Read the text input and load it into Python as a `Pair`
2. In each Scheme list, evaluate the operator (figure out if it's a `+`, `car`, etc.)
3. Recursively evaluate the operands (i.e. parameters) of the operation
4. Apply the operator to the operands and return the result

One of the challenges of designing interpreters is to represent the input in a way that the interpreter's language can understand. For example, since our Scheme interpreter is written in Python, we need to convert Scheme tokens to a Python representation. To achieve this, we will use the `Pair` object, which is essentially a Linked List that takes in `nil` instead of `Link.empty`.

As an example, `(list 1 2 3)` in Scheme can be converted to `Pair('list', Pair(1, Pair(2, Pair(3, nil))))`. This conversion is done in the Read step of the Read-Eval-Print loop. Note that nothing is evaluated in the Read step yet- everything is treated as just another token.

The following questions refer to the Scheme interpreter. Assume we're using the implementation seen in lecture and in the Scheme project.

1. What's the purpose of the read stage in a Read-Eval-Print Loop? For our Scheme interpreter, what does it take in, and what does it return?
2. What are the two components of the read stage? What do they do?
3. Write out the constructor for the Pair object the read stage creates with the input string `(define (foo x) (+ x 1))`
4. For the previous example, imagine we saved that Pair object to the variable `p`. How could we check that the expression is a `define` special form? How would we access the name of the function and the body of the function?

5. Circle or write the number of calls to `scheme_eval` and `scheme_apply` for the code below.

```
(if 1 (+ 2 3) (/ 1 0))
```

```
scheme_eval    1 3 4 6
```

```
scheme_apply   1 2 3 4
```

```
(or #f (and (+ 1 2) 'apple) (- 5 2))
```

```
scheme_eval    6 8 9 10
```

```
scheme_apply   1 2 3 4
```

```
(define (square x) (* x x))
```

```
(+ (square 3) (- 3 2))
```

```
scheme_eval    2 5 14 24
```

```
scheme_apply   1 2 3 4
```

```
(define (add x y) (+ x y))
```

```
(add (- 5 3) (or 0 2))
```

3 Scheme Challenge

1. Finish the functions `max` and `max-depth`. `max` takes in two numbers and returns the larger. Function `max-depth` takes in a list `lst` and returns the maximum depth of the list. In a nested scheme list, we define the depth as the number of scheme lists a sublist is nested within. A scheme list with no nested lists has a `max-depth` of 0.

```
;doctests
```

```
scm> (max 1 5)
```

```
5
```

```
scm> (max-depth '(1 2 3))
```

```
0
```

```
scm> (max-depth '(1 2 (3 (4) 5)))
```

```
2
```

```
scm> (max-depth '(0 (1 (2 (3 (4) 5) 6) 7))
```

```
4
```

```
(define (max x y) _____)
```

```
(define (max-depth lst)
```

```
  (define (helper lst curr)
```

```
    (cond
```

```
      ((_____) _____)
```

```
      ((_____) (max _____  
                    _____))
```

```
      (else (helper _____))
```

```
    )
```

```
  )
```

```
  (_____)
```

```
)
```

4 Past Exam Questions

- [Su19 Final Q7b](#)
- [Sp19 Final Q7a](#)