TAIL CALLS, MACROS, SCHEME CHALLENGE Solutions

COMPUTER SCIENCE MENTORS

April 12 - April 15, 2021

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)`? In Scheme, unlike in Python, we can use a method called **tail recursion**, which solves this problem by only using 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.

1. What is a tail call? What is a tail context? What is a tail recursive function?

   A tail call is a call expression in a tail context.
   A tail context is usually the final action of a procedure/function.
   A tail recursive function is a function where all its recursive calls are in tail contexts.

2. Why are tail calls useful for recursive functions?

   When a function is tail recursive, it can effectively discard all the past recursive frames and only keep the current frame in memory. This means we can use a constant amount of memory with recursion, and that we can deal with an unbounded number of tail calls with our Scheme interpreter.

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`.

`count-instance` returns the number of times `x` appears in `lst`.
It is not tail recursive. The call to `count-instance` is an arguments to a function call, so it will not be the final thing we do in every frame (we will have to apply + after evaluating it.)

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

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




)
```

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

5. Implement `filter`, 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 (lambda (x) (> x 2)) '(1 2 3 4 5))
(3 4 5)

(define (filter f lst)











)
```

```
(define (filter f lst)
  (define (filter-tail lst so-far)
     (cond ((null? lst) so-far)
           ((f (car lst)) (filter-tail (cdr lst)
                                       (append so-far (list (car lst)))))
           (else (filter-tail (cdr lst) so-far))))
  (filter-tail lst nil))
```

2 Macros

Macros Overview Whereas normal Scheme evaluation entails evaluating the operator, then evaluating the operands, before finally applying the operator on operands, macros evaluation involves three steps:

1. Evaluate the operator
2. Evaluate the body of the macro procedure without evaluating the operands
3. Evaluate the expression produced by the body and return the result.

Because the body is evaluated without evaluating the operands at first, macros are powerful and allow us to do more than scheme procedures, like implementing new special forms, control the order of evaluation, and more.

Quoting, Quasiquoting, Unquoting All Scheme expressions are lists except for atomic expressions like numbers and symbols; so call expressions and special forms are lists too; Example: (+ 1 2)

The `(quote expression)` special form, also denoted by a ´, simply returns `expression` - it does not evaluate it. This means we can write a Scheme expression and have the expression remain as an expression; if an expression is a call expression or special form, this means the expression will remain a list.

The `(quasiquote expression)` special form, `, has the same effect as `quote`, except that any expression within `expression` can be unquoted by preceding it with `,` or the `unquote` special form; any unquoted expression is evaluated, whereas everything else within `expression` is not, as normal. Quasiquote and unquote are often used in the body of macro procedures to selectively evaluate certain parts.

`(eval expression)` is a procedure that simply evaluates its argument. Note that since `eval` is a procedure, `expression` is evaluated first before applying `eval`.

Below is a simple example of a macro. Note that even though we pass in `(print 'hello)` as an operands, we don't evaluate the expression and print right away. Instead we first evaluate the body of the macro procedure, and afterwards we evaluate the expression produced by the macro.

```
(define-macro (twice expr)
(list 'begin expr expr))

scm> (twice (print 'hello))
hello
hello
```

CSM 61A Spring 2021: Ivan Penev and Joshua Baum, with
Jamie Ip, Kenneth Chi, Kevin Moy, Mathilde McKeever, Cyrus Bugwadia, Peter Zhang, and
Kunal Agarwal, Luke Liu, John So, Jemmy Zhou, Ruslana Yurtyn, Vivian Lu, Matt Au, Lily Yang, Mihira Patel, Jessica Yu, Aidan Tong, Matthew Guo, Nikhita Anumukonda, Laryn Qi, Grace Yi, Jennifer Huang, Cindy Lin, James Fletcher, Sean Sananikone, Abinaya Srikant, Uma Unni

1. What will Scheme output?

```
scm> (define x 6)
x
scm> (define y 1)
y
scm> '(x y a)
(x y a)
scm> `(,x ,y a)
(6 1 a)
scm> `(,x y a)
(6 y a)
scm> `(,(if (- 1 2) '+ '-) 1 2)
(+ 1 2)
scm> (eval `(,(if (- 1 2) '+ '-) 1 2))
3
scm> (define (add-expr a1 a2)
              (list '+ a1 a2))
add-expr
scm> (add-expr 3 4)
(+ 3 4)
scm> (eval (add-expr 3 4))
7
scm> (define-macro (add-macro a1 a2)
            (list '+ a1 a2))
add-macro
scm> (add-macro 3 4)
7
```

2. Implement `if-macro`, which behaves similarly to the `if` special form in Scheme but has some additional properties. Here's how the `if-macro` is called:
   `if <cond1> <expr1> elif <cond2> <expr2> else <expr3>`
   If cond1 evaluates to a truth-y value, expr1 is evaluated and returned. Otherwise, if cond2 evaluates to a truth-y value, expr2 is evaluated and returned. If neither condition is true, expr3 is evaluted and returned.

```
;Doctests
scm> (if-macro (= 1 0) 1 elif (= 1 1) 2 else 3)
2
scm> (if-macro (= 1 1) 1 elif (= 2 2) 2 else 3)
1
scm> (if-macro (= 1 0) (/ 1 0) elif (= 2 0) (/ 1 0) else 3)
3

(define-macro (if-macro cond1 expr1 elif cond2 expr2 else
    expr3)







)
```

```
(define-macro (if-macro cond1 expr1 elif cond2 expr2 else
    expr3)
      (list 'cond (list cond1 expr1)
                  (list cond2 expr2)
                  (list 'else expr3)))
```

Alternate solution with nested ifs:

```
(define-macro (if-macro cond1 expr1 elif cond2 expr2 else
    expr3)
      (list 'if cond1 expr1 (list 'if cond2 expr2 expr3)))
```

Alternate solution with quasiquoting:

```
(define-macro (if-macro cond1 expr1 elif cond2 expr2 else
    expr3)
      `(cond (,cond1 ,expr1)
             (,cond2 ,expr2)
             (else ,expr3)))
```

3. Could we have implemented `if-macro` using a function instead of a macro? Why or why not?

   Without using macros, the inputs would be evaluated when we evaluated the function call. This is problematic for two reasons:
   First, we only want to evaluate the expressions under certain conditions. If cond1 was false, we would not want to evaluate expr1. This might lead to errors!
   Secondly, some of the inputs to the call would be names which have no binding in the global frame. Elif, for example, is not supposed to be interpreted as a name but rather as a symbol. This would cause our code to error if we ran it as is!
   Of course, we could have written out a `cond` or nested `if` expression instead of defining an `if-macro`. But the syntax for `if-macro` is more familiar, which is why we might want to do something like this!

4. Implement `apply-twice`, which is a macro that takes in a call expression with a single argument. It should return the result of applying the operator to the operand twice.

```
;Doctests
scm> (define add-one (lambda (x) (+ x 1)))
add-one
scm> (apply-twice (add-one 1))
3
scm> (apply-twice (print 'hi))
hi
undefined
```

```
(define-macro (apply-twice call-expr)

    `(let ((operator ______________________________)

              (operand ______________________________))

              (________________________________________________________)))
```

```
(define-macro (apply-twice call-expr)
    `(let ((operator ,(car call-expr))
              (operand ,(car (cdr call-expr))))
           (operator (operator operand))))
```

CSM 61A Spring 2021: Ivan Penev and Joshua Baum, with
Jamie Ip, Kenneth Chi, Kevin Moy, Mathilde McKeever, Cyrus Bugwadia, Peter Zhang, and
Kunal Agarwal, Luke Liu, John So, Jemmy Zhou, Ruslana Yurtyn, Vivian Lu, Matt Au, Lily Yang, Mihira Patel, Jessica Yu, Aidan Tong, Matthew Guo, Nikhita Anumukonda, Laryn Qi, Grace Yi, Jennifer Huang, Cindy Lin, James Fletcher, Sean Sananikone, Abinaya Srikant, Uma Unni

3 Scheme Challenge

1. Suppose Isabelle bought turnips from the Stalk Market and has stored them in random amounts among an ordered sequence of boxes. By the magic of time travel, Isabelle's friend Tom Nook can fast-forward one week into the future and determine exactly how many of Isabelle's turnips will rot over the week and have to be discarded.

   Assuming that boxes of turnips will rot in order, i.e. all of box 1's turnips will rot before any of box 2's turnips, help Isabelle determine which turnips will still be fresh by week's end. Specifically, fill in `decay`, which takes in a list of positive integers `boxes`, which represents how many turnips are in each box, and a positive integer `rotten` representing the number of turnips that will rot, and returns a list of non-negative integers that represents how many fresh turnips will remain in each box.

```
; doctests
scm> (define a '(1 6 3 4))
a
scm> (decay a 1)
(0 6 3 4)
scm> (decay a 5)
(0 2 3 4)
scm> (decay a 9)
(0 0 1 4)
scm> (decay a 1000)
(0 0 0 0)

(define (decay boxes rotten)




)
```

```
(define (decay boxes rotten)
    (cond
        ((null? boxes) nil)
        ((< rotten (car boxes)) (cons (- (car boxes) rotten)
           (cdr boxes)))
        (else (cons 0 (decay (cdr boxes) (- rotten (car
           boxes)))))
    )
)
```

2. 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 __________________________))
        )
    )
    (____________________________)
)
```

```
(define (max x y) (if (> x y) x y))

(define (max-depth lst)
    (define (helper lst curr)
            (cond
              ((null? lst) curr)
              ((pair? (car lst)) (max (helper (car lst)
                                              (+ 1 curr))
                                      (helper (cdr lst) curr)))
              (else (helper (cdr lst) curr))
            )
      )
    (helper lst 0)
)
```

CSM 61A Spring 2021: Ivan Penev and Joshua Baum, with
Jamie Ip, Kenneth Chi, Kevin Moy, Mathilde McKeever, Cyrus Bugwadia, Peter Zhang, and
Kunal Agarwal, Luke Liu, John So, Jemmy Zhou, Ruslana Yurtyn, Vivian Lu, Matt Au, Lily Yang, Mihira Patel, Jessica Yu, Aidan Tong, Matthew Guo, Nikhita Anumukonda, Laryn Qi, Grace Yi, Jennifer Huang, Cindy Lin, James Fletcher, Sean Sananikone, Abinaya Srikant, Uma Unni