CS 61A
Interpreters, Tail Calls, Macros, Streams, Iterators
Spring 2019

Guerrilla Section 5: April 20, 2019 Solutions

1 Interpreters

Questions

1.1 Determine the number of calls to **scheme_eval** and the number of calls to **scheme_apply** for the following expressions.

```
> (+ 1 2)
3
```

4 calls to eval:
1. Evaluate entire expression
2. Evaluate +
3. Evaluate 1
4. Evaluate 2

1 call to apply:
1. Apply + to 1 and 2

```
> (if 1 (+ 2 3) (/ 1 0))
5
```

6 calls to eval:
1. Evaluate entire expression
2. Evaluate predicate 1
3. Since 1 is true, evaluate the entire sub-expression (+ 2 3)
4. Evaluate +
5. Evaluate 2
6. Evaluate 3

1 call to apply:
1. Apply + to 2 and 3

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

8 calls to eval:
1. Evaluate entire expression
2. Evaluate false
3. Evaluate entire sub-expression (and (+ 1 2) 'apple)

4. Evaluate entire sub-expression (+ 1 2)
5. Evaluate +
6. Evaluate 1
7. Evaluate 2
8. Evaluate 'apple

Since the and expression evaluates to true, we short circuit here.
1 call to apply:
1. Apply + to 2 and 3

```
> (define (add x y) (+ x y))
add
> (add (- 5 3) (or 0 2))
2
```

13 calls to eval:
1. Evaluate entire define expression
2. Evaluate entire (add ...) expression
3. Evaluate add operator
4. Evaluate entire (- 5 3) sub-expression
5. Evaluate -
6. Evaluate 5
7. Evaluate 3
8. Evaluate entire (or 0 2) sub-expression
9. Evaluate 0 (and short circuit, since 0 is truthy in Scheme
10. Evaluate (+ x y) (after applying add and entering the body of the add function)
11. Evaluate +
12. Evaluate x
13. Evaluate y

2 call to apply:
1. Apply - to 5 and 3
2. Apply + to -2 and 0

2 Tail Calls

Questions

2.1 For the following procedures, determine whether or not they are tail recursive. If they are not, write why not and rewrite the function to be tail recursive on the right.

```
; Multiplies x by y
(define (mult x y)
    (if (= 0 y)
        0
        (+ x (mult x (- y 1)))))
```

```
(define (mult x y)
    (define (helper x y total)
        (if (= 0 y)
            total
            (helper x (- y 1) (+ total x))))
    (helper x y 0))
```

Not tail recursive: after evaluating the recursive call, we still need to apply +, so evaluating the recursive call is not the last thing we do in the frame.

```
; Always evaluates to true
; assume n is positive
(define (true1 n)
    (if (= n 0)
        #t
        (and #t (true1 (- n 1)))))
```

Tail recursive: the recursive call to true1 is the final sub-expression of the ‘and‘ special form. Therefore, we will not need to perform any additional work after getting the result of the recursive call.

```
; Always evaluates to true
; assume n is positive
(define (true2 n)
    (if (= n 0)
        #t
        (or (true2 (- n 1)) #f)))
```

```
(define (true2 n)
    (if (= n 0)
        #t
        (true2 (- n 1))))
```

Not tail recursive: the recursive call to true2 is not the final sub-expression of the ‘or‘ special form. Even though it will always evaluate to ‘true‘ and short-circuit, the

interpreter does not take that into account when determining whether to evaluate it in a tail context or not.

```
; Returns true if x is in lst
(define (contains lst x)
    (cond
        ((null? lst) #f)
        ((equal? (car lst) x) #t)
        ((contains (cdr lst) x) #t)
        (else #f)))
```

```
(define (contains lst x)
    (cond ((null? lst) #f)
        ((equal? (car lst) x) #t)
        (else (contains (cdr lst) x))))
```

Not tail recursive: the recursive call to contains is in a predicate sub-expression. That means we will have to evaluate another expression if it evaluates to true, so it is not the final thing we evaluate.

2.2 Tail recursively implement **sum-satisfied-k** which, given an input list **lst**, a predicate procedure **f** which takes in one argument, and an integer **k**, will return the sum of the first **k** elements that satisfy **f**. If there are not **k** such elements, return 0.

```
; Doctests
scm> (define lst `(1 2 3 4 5 6))
scm> (sum-satisfied-k lst even? 2)  ; 2 + 4
6
scm> (sum-satisfied-k lst (lambda (x) (= 0 (modulo x 3))) 10)
0
scm> (sum-satisfied-k lst (lambda (x) #t) 0)
0

(define (sum-satisfied-k lst f k)









)
```

```
(define (sum-satisfied-k lst f k)
    (define (sum-helper lst k total)
        (cond ((= 0 k) total)
            ((null? lst) 0)
         ((f (car lst)) (sum-helper (cdr lst) (- k 1) (+ total (car lst))))
            (else (sum-helper (cdr lst) k total))))
(sum-helper lst k 0))
```

2.3 Tail-recursively implement **remove-range** which, given one input list **lst**, and two nonnegative integers **i** and **j**, returns a new list containing the elements of **lst** in order, without the elements from index **i** to index **j** inclusive. For example, given the list (0 1 2 3 4), with $\mathbf{i} = 1$ and $\mathbf{j} = 3$, we would return the list (0 4). You may assume $\mathbf{j} > \mathbf{i}$, and **j** is less than the length of the list. (Hint: you may want to use the built-in **append** function, which returns the result of appending the items of all lists in order into a single well-formed list.)

```
; Doctests
scm> (remove-range (0 1 2 3 4) 1 3)
(0 4)

(define (remove-range lst i j)
```

```
(define (remove-range lst i j)
    (define (remove-tail lst index so-far)
        (cond ((> index j) (append so-far lst))
            ((>= index i) (remove-tail (cdr lst) (+ index 1) so-far))
        (else (remove-tail (cdr lst) (+ index 1) (append so-far (list (car lst)))))))
    (remove-tail lst 0 nil)))
```

3 Macros

Questions

3.1 What will Scheme display? If you think it errors, write Error

```
> (define-macro (doierror) (/ 1 0))
```

```
doierror
```

```
> (doierror)
```

```
Error
```

```
> (define x 5)
```

```
x
```

```
>(define-macro (evaller y) (list (list 'lambda '(x) 'x) y))
```

```
evaller
```

```
> (evaller 2)
```

```
2
```

3.2 Consider a new special form, **when**, that has the following structure:

```
(when <condition> <expr1> <expr2> <expr3> ... )
```

If the condition is not false (a truthy expression), all the subsequent operands are evaluated in order and the value of the last expression is returned. Otherwise, the entire **when** expression evaluates to **okay**.

```
scm> (when (= 1 0)(/1 0) 'error)
okay
scm> (when (= 1 1) (print 6) (print 1) 'a)
6
1
a
```

Create this new special form using a macro. Recall that putting a dot before the last formal parameter allows you to pass any number of arguments to a procedure, a list of which will be bound to the parameter, similar to (*args) in Python.

```
; implement when without using quasiquotes
(define-macro (when condition . exprs)
    (list 'if _____________________________________________________________________________))

; implement when using quasiquotes
(define-macro (when condition . exprs)
    `(if _____________________________________________________________________________))
```

```
; without quasiquotes
(define-macro (when condition . exprs)
    (list 'if condition (cons 'begin exprs) ''okay))

; with quasiquotes
(define-macro (when condition . exprs)
    `(if ,condition ,(cons 'begin exprs) 'okay))
```

4 Streams

Questions

4.1 What Would Scheme Display?

```
> (define a (cons-stream 4 (cons-stream 6 (cons-stream 8 a))))
> (car a)

4
> (cdr a)

[promise (not forced)]
> (cdr-stream a)

(6 . [promise (not forced)])
> (define b (cons-stream 10 a))
> (cdr b)

[promise (not forced)]
> (cdr-stream b)

(4 . [promise (forced)])
> (define c (cons-stream 3 (cons-stream 6)))
> (cdr-stream c)

Error: too few operands in form
```

4.2 Write a function **merge** that takes in two sorted infinite streams and returns a new infinite stream containing all the elements from both streams, in sorted order.

```
(define (merge s1 s2)




)
```

```
(define (merge s1 s2)
(if (null? s1)
s2
(if (<= (car s1) (car s2))
(cons-stream (car s1) (merge (cdr-stream s1) s2))
(cons-stream (car s2) (merge s1 (cdr-stream s2)))
)
)
)

; Alternate solution
(define (merge s1 s2)
(if (null? s1)
s2
(if (<= (car s1) (car s2))
(cons-stream (car s1) (merge (cdr-stream s1) s2))
(merge s2 s1)
)
)
)
```

5 Iterators

Questions

5.1 What is the definition of an iterable? What is the definition of an iterator? What is the definition of a generator?

An iterable is any object that can be passed to the built-in iter function. In other words, an iterable is any object that can produce iterators.
An iterator is an object that provides sequential access to values one by one. Its contents can be accessed through the built-in next function, and it will signal there are no more values available with a StopIteration exception when next is called.
A generator object is an iterator, but it is created in a special way – generator functions are defined as a function that yields its values instead of returning them. When generator functions are called, they return a generator object, which can then be used as an iterator.

5.2 What Would Python Display?

```
>>> def g(n):
        while n > 0:
            if n % 2 == 0:
                yield n
            else:
                print('odd')
            n -= 1
>>> t = g(4)
>>> t

<Generator Object>
>>> next(t)

4
>>> n

NameError: name 'n' is not defined
>>> t = g(next(t) + 5)

odd
>>> next(t)

odd
    6
```

5.3 Write a generator function textbfgen_inf that returns a generator which yields all the numbers in the provided list one by one in an infinite loop.

```
>>> t = gen_inf([3, 4, 5])
>>> next(t)
3
>>> next(t)
4
>>> next(t)
5
>>> next(t)
3
>>> next(t)
4
def gen_inf(lst):


#solution 1
def gen_inf(lst):
    while True:
        for elem in lst:
            yield elem
#solution 2
def gen_inf(lst):
    while True:
        yield from iter(lst)
```

5.4 Write a function **nested_gen** which, when given a nested list of iterables (including generators) **lst**, will return a generator that yields all elements nested within **lst** in order. Assume you have already implemented **is_iter**, which takes in one argument and returns **True** if the passed in value is an iterable and **False** if it is not.

```
def nested_gen(lst):
    '''
    >>> a = [1, 2, 3]
        >>> def g(lst):
        >>>     for i in lst:
        >>>             yield i
        >>> b = g([10, 11, 12])
        >>> c = g([b])
        >>> lst = [a, c, [[[2]]]]
        >>> list(nested_gen(lst))
        [1, 2, 3, 10, 11, 12, 2]
    '''


    for elem in lst:
        if is_iter(elem):
            yield from nested_gen(elem)
        else:
            yield elem
```