MACROS, TAIL RECURSION AND INTERPRETERS

COMPUTER SCIENCE MENTORS CS 61A

April 9 to April 11, 2018

1 Let in Scheme

1. **let** is a special form in Scheme which allows you to create local bindings. Consider the example

```
(let ((x 1)) (+ x 1))
```

Here, we assign \times to 1, and then evaluate the expression (+ \times 1) using that binding, returning 2. However, outside of this expression, \times would not be bound to anything.

Each let special form has a corresponding lambda equivalent. The equivalent lambda expression for the above example is

```
((lambda (x) (+ x 1)) 1)
```

The following line of code does not work. Why? Write the lambda equivalent of the let expressions.

```
(let ((foo 3)
(bar (+ foo 2)))
(+ foo bar))
```

1. What will Scheme output?

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 evaluated 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

Schene code

(Cond (Cond | expr)

(e(se exp-3))

Marro List

(list cond) expr

(list cont expr)

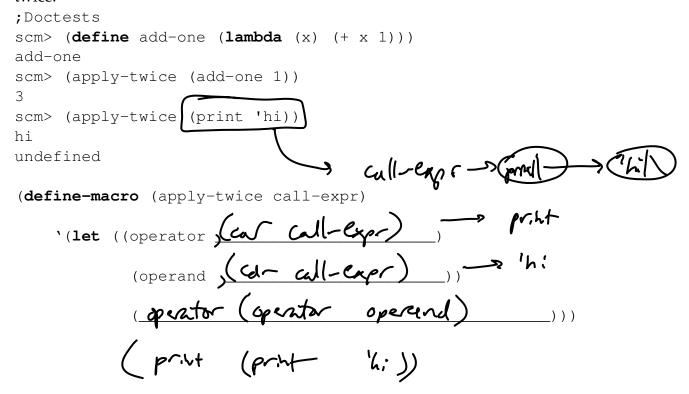
Blendsquite

(Gnd (, Gnd , exprl) (, cond , exprl) (e(sp. , exprl)

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

No, using afunction, he would always be evaluately each agreent once letter passing in +> time call.

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.



Tail Recursion

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

func all in tail untext 2. Why are tail calls useful for recursive functions?

O(1) space

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 sum-list with $lst = (1 \ 2 \ 1)$ and x = 1.

No (+ 1 (court-instance (cdr 1st) x))

4. Rewrite count-instance to be tail recursive.

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.

```
cons (filter (lambda (x) (> x 2)) '(1 2 3 4 5))

(define (filter f lst)

(define (filter-tr f lst retlst)

(cond ((nuli? 1st) retlst)

((f (car 1st)) (filter-tr f (drlst) (aprend retlst)

(else (filter-tr f (car 1st) retlst))

(filter-tr f lst nil)
```

Interpreters

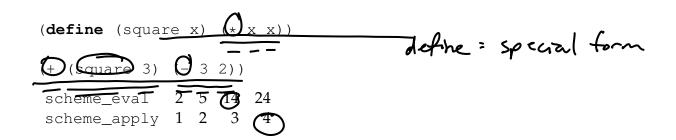
1. Circle the number of calls to scheme_eval and scheme_apply for the code below.

scheme_eval
$$\begin{bmatrix} 1 & 3 & 4 \\ 3 & 6 \end{bmatrix}$$
 scheme_apply $\begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix}$

2. Write the number of calls to scheme_eval and scheme_apply for the code below.

or, and, quote: spend for

if : spend from



or = special form

13 eval 3 aprhy