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 `x` to `1`, and then evaluate the expression `(+ x 1)` using that binding, returning `2`. However, outside of this expression, `x` 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))
```

Solution: The above function will error because it is equivalent to:

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

In other words, foo has not been defined in the global frame. When bar is being assigned to `(+ foo 2)`, it will error. The assignment of foo to 3 happens in the lambda?s frame when it's called, not the global frame (this is relevant to the Scheme project – when the interpreter sees `lambda`, it will call a function to start a new frame).

If we had the line `(define foo 3)` before the call to let, then it would return 8, because within let, foo would be 3 and bar would be `(+ 3 2)`, since it would use the foo in the Global frame.

2 Macros

1. What will Scheme output?

```
scm> (define x 6)
```

Solution: `x`

```
scm> (define y 1)
```

Solution: `y`

```
scm> '(x y a)
```

Solution: `(x y a)`

```
scm> `(,x ,y a)
```

Solution: `(6 1 a)`

```
scm> `(,x y a)
```

Solution: `(6 y a)`

```
scm> `(,(if (- 1 2) '+ '-) 1 2)
```

Solution: `(+ 1 2)`

```
scm> (eval `(,(if (- 1 2) '+ '-) 1 2))
```

Solution: `3`

```
scm> (define (add-expr a1 a2)
             (list '+ a1 a2))
```

Solution: `add-expr`

```
scm> (add-expr 3 4)
```

Solution: `(+ 3 4)`

```
scm> (eval (add-expr 3 4))
```

Solution: `7`

```
scm> (define-macro (add-macro a1 a2)
            (list '+ a1 a2))
```

Solution: `add-macro`

```
scm> (add-macro 3 4)
```

Solution: `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)









)
```

Solution:
```
(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?

Solution: 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 ____________________________))

            (________________________________________________________)))
```

Solution:

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

3 Tail Recursion

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

Solution: 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 where all the recursive calls of the function are in tail contexts.

An ordinary recursive function is like building up a long chain of domino pieces, then knocking down the last one. A tai l recursive function is like putting a domino piece up, knocking it down, putting a domino piece up again, knocking it d own again, and so on. This metaphor helps explain why tail calls can be done in constant space, whereas ordinary recursi ve calls need space linear to the number of frames (in the metaphor, domino pieces are equivalent to frames).

2. Why are tail calls useful for recursive functions?

Solution: When a function is tail recursive, it can effectively discard all the past recursive frames and only keep the current fr ame 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 sum-list with `lst = (1 2 1)` and `x = 1`.

Solution: `count-instance` returns the number of time `x` appears in `lst`. It is not tail recursive. The call to `count-instance` appears as one of the 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.

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



)
```

Solution:

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






)
```

Solution:

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

4 Interpreters

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

```
(+ 1 2)
```

`scheme_eval`	1	3	4	6
`scheme_apply`	1	2	3	4

Solution: 4 `scheme_eval`, 1 `scheme_apply`.

2. 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

Solution: 6 `scheme_eval`, 1 `scheme_apply`.

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

`scheme_eval`	6	8	9	10
`scheme_apply`	1	2	3	4

Solution: 8 `scheme_eval`, 1 `scheme_apply`.

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

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

`scheme_eval`	2	5	14	24
`scheme_apply`	1	2	3	4

Solution: 14 `scheme_eval`, 4 `scheme_apply`.

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

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

Solution: 13 `scheme_eval`, 3 `scheme_apply`.