PROGRAMS AS DATA & MACROS

COMPUTER SCIENCE MENTORS 61A

April 15–April 19, 2024

1 Macros

Up to now, all of our programs have taken data—integers, strings, numbers, lists, and more—and manipulated this data to accomplish a wide variety of tasks. What if computer code itself could serve as the data used and produced by programs? What if we could treat **programs as data**?

We've already seen something similar when we built higher order functions. Functions can be thought of as "bundles of code," and HOFs allowed us to treat this code as input and output to other functions. For example, the `twice` function below takes in a zero-argument function `f` and calls `f` twice.

```
def twice(f):
    f()
    f()

>>> twice(lambda: print(5))
5
5
```

```
(define (twice f)
    (f)
    (f))

scm> (twice (lambda () (print 5)))
5
5
```

Beyond HOFs, Scheme gives us many more tools for the treatment of programs as data.

1.1 Quotation

Recall that Scheme is a list processing language. The call expression `(+ 1 2)` is a list literally consisting of the symbol + and the numbers `1` and `2`. When this list is provided the interpreter, the appropriate evaluation rules are followed and the expression evaluates to `3`. Similarly, when we provide the list `(1 2 3)` to the Scheme interpreter, it attempts to evaluate the list as a call expression and encounters an error. In Scheme, there is no distinction between lists and call expressions/special forms.

Since Scheme programs are essentially stored as a long list, if we are to use programs as input to other programs, we need a way to prevent them from being evaluated while we manipulate them. The `quote` special form, also denoted by an apostrophe ´, which simply returns its unevaluated operand:

```
scm> '(+ 1 2)
(+ 1 2)
```

Created by Alyssa Smith, Esther Shen

```
scm> (list '(if #t (\ 1 0) 3)(+ 2 3))
((if #t (\ 1 0) 3) 5)
```

Quotation is a "protective shell" that prevents the immediately following expression from being evaluated as it passes through the interpreter.

On the other hand, **`eval`** is a procedure that simply evaluates its argument. Note that since **`eval`** is a procedure, its argument is evaluated first before applying **`eval`**. Whereas quotation prevents evaluation, **`eval`** evaluates things another time.

```
scm> (eval '(+ 1 2))
3
scm> (eval (list 1 2 3))
Error: int is not callable
```

Quotation allows us to begin taking code as input. For example, the following version of `twice` takes in an expression and evaluates it twice:

```
(define (twice expr)
    (eval expr)
    (eval expr))

scm> (twice '(print 5))
5
5
```

1.2 Quasiquotation

The quasiquote special form, denoted with a backtick `` ` ``, has the same effect as `'`, except that any subexpressions can be "unquoted" by preceding them with a comma `,`. Any unquoted subexpression is evaluated as normal, whereas everything else is left unevaluated.

```
(define (cool-string tens-digit ones-digit letter)
    (I love ,tens_digit ,ones_digit ,letter))

scm> (cool-string 6 1 'a)
(i love 6 1 a)
```

This is very similar to f-string behavior in Python:

```
def cool_string(tens_digit, ones_digit, letter):
    return f"I love {tens_digit}{ones_digit}{letter}"

>>> cool_string(6, 1, "a")
'I love 61a'
```

The analogy is summed up by the following:

- Quotation `'...` in Scheme is like strings `"..."` in Python
- Quasiquotation `` `... `` in Scheme is like f-strings in Python `f"..."`
- Unquotation `,...` in Schemes is like replacement fields `{...}` in Python

1.3 Macros

A call expression in Scheme is evaluated by evaluating the operator, then evaluating the operands, before finally applying the operator to the operands. Because the parameters of a Scheme procedure are evaluated before the body of the procedure is evaluated, we say that procedures operate on values.

In Scheme, a **macro** is similar to a procedure, but it operates on expressions rather than value. Thus the input to macros is code that we can manipulate as data. Macro evaluation involves three steps:

1. Evaluate the operator to a macro procedure.
2. Apply the macro procedure to the *unevaluated* operands
3. Evaluate the expression produced by the macro procedure in the same frame it was called in and return the result.

This may sound overwhelming at first, but just remember that the key difference between macro procedures and regular procedures are that 1) in macros, operands are not evaluated and 2) after the body of the macro produces an expression, the expression is automatically evaluated one more time.

New macros are defined using the special form **`define-macro`**. Below is an example of a macro `twice` that evaluates a given expression twice.

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

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

When `twice` is called, the unevaluated expression `'(print 'hello)` is bound to the symbol `expr`. The body of the macro is executed as normal, producing the expression `(begin (print 'hello)(print 'hello))`. Finally, this expression is evaluated in the global frame, and the macro call prints `hello` twice.

Note that even though we pass in `(print 'hello)` as an operand, we don't evaluate the expression and print right away. Because the body is evaluated without evaluating the operands at first, macros allow us to implement new special forms, control the order of evaluation, and more.

1. What will Scheme output?

```
scm> (define x 6)


scm> (define y 1)


scm> '(x y a)


scm> `(,x ,y a)


scm> `(,x y a)


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


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


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


scm> (add-expr 3 4)


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


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


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

2. The built-in `apply` procedure in Scheme applies a procedure to a given list of arguments. For example, `(apply f '(1 2 3))` is equivalent to `(f 1 2 3)`. Write a macro procedure `meta-apply`, which is similar to `apply`, except that it works not only for procedures, but also for macros and special forms. That is, `(meta-apply operator (operand1 ... operandN))` should be equivalent to `(operator operand1 ... operandN)` for any operator and operands. See doctests for examples.

```
; Doctests
scm> (meta-apply + (1 2))
3
scm> (meta-apply or (#t (/ 1 0) #f))
#t
(define-macro (meta-apply operator operands)


)
```

3. NAND (not and) is a logical operation that returns false if all of its operands are true, and true otherwise. That is, it returns the opposite of AND. Implement the `nand` macro procedure below, which takes in a list of expressions and returns the NAND of their values. Similar to **`and`**, `nand` should short circuit and return true as soon as it encounters a false operand, evaluating from left to right.

   Hint: You may use `meta-apply` in your implementation.

```
;Doctests
scm> (nand (#t #t #t #t #t #t))
#f
scm> (nand (#t #f #t))
#t
scm> (nand (#f (/ 1 0)))
#t
(define-macro (nand operands))


)
```

4. Implement the macro function `combine-num`, which takes in an unquoted list of positive integers and returns a number whose digits are the items of the list in reverse order.

```
;Doctests
scm> (combine-num (1 2))
; 21
scm> (combine-num (2 5 3 5))
; 5352
scm> (combine-num (1))
; 1
scm> (+ (combine-num (1 2 3 4)) 5)
; 4326      # (4321 + 5)


(define-macro (combine-num lst)




)
```

5. 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)
                ____________________________
)
```

6. Recall that in Scheme, all expressions are truthy except for the boolean `#f`. In contrast, Python considers empty lists, `None`, and `0` falsy as well. Implement the macro `python-if`, which acts just like the **`if`** special form, but it treats empty lists, `undefined`, `0`, and `#f` as falsy and all other values as truthy.

```
;Doctests
scm> (python-if (- 1 1) (/ 1 0) 1)
1
scm> (python-if (not #t) (/ 1 0) 2)
2
scm> (python-if (cdr '(1)) (/ 1 0) 3)
3
scm> (python-if (print 4) (/ 1 0) 5)
4
5
scm> (python-if (- 4 3) 6 (/ 1 0))
6
```

```
(define-macro (python-if pred if-true if-false)









)
```

7. Write a macro procedure `censor`, which takes in an expression `expr` and a symbol `phrase`. If `expr` does not contain any instance of `phrase`, then `censor` simply evaluates `expr`. However, if `expr` does contain an instance of the censored phrase, the symbol `censored` is returned and the expression is not evaluated.

```
;Doctests
scm> (censor ((lambda (stanford tree) (+ stanford tree)) 4 5) stanford)
censored
scm> (censor ((lambda (stanford tree) (+ stanford tree)) 4 5) tree)
censored
scm> (censor ((lambda (stanford tree) (+ stanford tree)) 4 5) ree)
9

(define-macro (censor expr phrase)
    (define (contains-phrase expr)




    )
    (if _____________________________________________

        _____________________________________________

        _____________________________________________))
```