CS 61A
Fall 2022

Programs as Data
Discussion 11: November 9, 2022

Programs as Data

So far in Python, we've created functions and expressions that will return a value or an object when we evaluate them.

```
>>> print(print(5))
5
None
>>> (lambda f, a: f(a))(lambda x: x * x, 3)
9
```

But what if there was a way to treat our code not as functions or expressions, but rather as data that we can manipulate. For example, imagine if we wanted to write a function in Python that would create and evaluate the following line of code:

```
[<expr> for i in range(5)]
```

Where we could pass in any arbitrary expression `<expr>`, that would then be evaluated 5 times and have the results listed.

In the example above, `<expr>` is not necessarily an object, but could be any piece of code, such as `print("Hello")`, `[j ** 2 for j in range(i)]`, or `i >= 5`. For these expressions it is important to our problem statement that they not be evaluated until *after* they have been inserted into our list comprehension, either because they have side effects that will be apparent in the global frame, or because they depend on `i` in some way (there can be other reasons too!). So, we need a procedure that, instead of manipulating objects, manipulates code itself!.

Let's see what happens if we try to solve this problem with a traditional function.

```
def list_5(expr):
    return [expr for i in range(5)]

>>> lst = list_5(print(10))
10
>>> lst
[None, None, None, None, None]
```

This isn't quite what we want. Because of Python evaluation rules, instead of evaluating a list of 5 `print(10)` statements, our `expr` was evaluated before the function was ever called, meaning 10 was only printed once.

The issue here is order of evaluation—we don't want our expression parameter to be evaluated until *after* it is inserted into our list comprehension. To enforce the order of evaluation we want, we can write by returning a **string** representing the return expression, and **evaluating** this string after it has been returned.

```
def list_5(expr):
    return f"[{expr} for i  in range(5)]"

>>> list_5("print(10)")
"[print(10) for i in range(5))]"
>>> lst = eval(list_5("print(10)"))
10
10
10
10
10
>>> lst
[None, None, None, None, None]
```

Now we see the number 10 printed 5 times as a side effect of our function, just like we want! We circumvented Python's evaluate-operands-before-evaluating-function body rule by passing in our desired expression as a string. Then, after we constructed our list comprehension in string form and we've returned it, we used the `eval` function to force evaluation of the output after the last step in the execution of this function. We were able to write code that took in code as a parameter and restructured it in the form of new code!

Although this is a cool hack we can do with Python, its usage is unfortunately rather limited. The `eval` function will throw a `SyntaxError` if it is passed any `compound` blocks such as `if: ...`, `while: ...`, `for: ...`, etc. (this does not include list comprehensions, or one-line if-statements such as `1 if a > b else 0`.)

Q1: If Program Python

Let's see an example of using programs as data that would allow us to circumvent normal python behavior.

Consider this implementation of a function that is supposed to do the same thing as a one-line `if` statement: `[on_true] if [expression] else [on_false]`

```
def if_function(condition, true_result, false_result):
    """Returns true_result if condition is a true value, and
    false_result otherwise.

    >>> if_function(True, 2, 3)
    2
    >>> if_function(False, 2, 3)
    3
    """
    if condition:
        return true_result
    else:
        return false_result
```

Although our function seems like it would behave as expected, it does not *short circuit* properly. For example, `if_function(True, 2, 1 / 0)`, `if_function` will error since Python first evaluates all the operands before invoking the function. That is, Python will execute the operand `1 / 0` and error before running the body of `if_function`.

Ideally, our function should short-circuit before it reaches `false_result` and only return `true_result` if `condition` is evaluated to be truth-y.

To enforce the order of evaluation we want, we can use programs as data! Let's create an alternate function called `if_program` that takes in three arguments:

1. `condition`: a string that will evaluate to a truth-y or false-y value
2. `true_result`: a string that will be evaluated and returned if `condition` is truth-y
3. `false_result`: a string that will be evaluated and returned if `condition` is false-y.

and returns a **string** that, when evaluated, will return the result of this `if` function.

`if_program` should **only** evaluate `true_result` if `condition` is truth-y, and will only evaluate `false_result` if `condition` is false-y.

Hint: You can write a one-line if statement with the following syntax: `<value_when_true> if <condition> else <value_when_false>`

Hint: Using f-Strings might make this problem easier too!

```
def if_program(condition, true_result, false_result):
    """
    >>> eval(if_program("True", "3", "4"))
    3
    >>> eval(if_program("0", "'if true'", "'if false'"))
    'if false'
    >>> eval(if_program("1", "print('true')", "print('false')"))
    true
    >>> eval(if_program("print('condition')", "print('true_result')", "print('
    false_result')"))
    condition
    false_result
    """
    return f"{true_result} if {condition} else {false_result}"
```

Scheme Programs as Data

Now let's use Scheme to build programs as data!

Recall that all Scheme programs are made up of expressions. There are two types of expressions: primitive (a.k.a atomic) expressions and combinations.

- Primitive/atomic expression examples: `#f`, `1.7`, `+`
- Combinations examples: `(factorial 10)`, `(/ 8 3)`, `(not #f)`

Importantly, Scheme stores any non-primitive expression as a Scheme list! This means we can build lists to represent our programs as data rather than using strings.

For example, `(list '+ 2 2)` results in the list `(+ 2 2)`. If we then call `eval` on this list, it will be evaluated as a combination *expression*, which will return 4!

```
scm> (define expr (list '+ 2 2))
expr
scm> expr
(+ 2 2)
scm> (eval expr)
4
```

The `eval` procedure forces evaluation of a given expression in the current environment. `eval` takes in one operand. That operand is first evaluated to a value, and then `eval` evaluates *that* value as another expression. In the line `(eval expr)`, `expr` is first evaluated to the list `(+ 2 2)`, and then `eval` evaluates that list as an expression to get `4`.

Since a quote suppresses evaluation, calling `eval` on a quoted expression `(quote expr)` will evaluate the expression `expr`.

```
scm> (define a '(1 2 3))
a
scm> (quote a) ; equivalently, 'a
a
scm> (eval (quote a))
(1 2 3)
```

Quasiquotation

The normal quote `'` and the quasiquote `` ` `` are both valid ways to quote an expression. However, the quasiquoted expression can be *unquoted* with the "unquote" `,` (represented by a comma). When a term in a quasiquoted expression is *unquoted*, the unquoted term is *evaluated*. Quasiquotation will be very helpful for helping us create lists that represent expressions.

```
scm> (define a 5)
a
scm> (define b 3)
b
scm> `(* a b)
(* a b)
scm> `(* a ,b)
(* a 3)
scm> '(* a ,b)
(* a (unquote b))
```

Q2: Writing Quasiquote Expressions

Given that the following code has been executed in your Scheme environment:

```
scm> (define a +)
a
scm> (define b 1)
b
scm> (define c 2)
c
```

Write out the expressions **using quasiquote, unquote, a, b, c, and parentheses** to fill in the blanks that would lead to each of the following being displayed:

For example,

```
scm> ___________
(a b c)
```

Answer:

```
`(a b c)
```

a) Fill in the following blank:

```
scm> __________________
(a 1 c)
```

```
`(a ,b c)
```

b) Fill in the following blank:

```
scm> __________________
3
```

```
(a b c)
```

c) Fill in the following blank:

```
scm> __________________
(a (b 1) c)
```

```
`(a (b ,b) c)
```

d) Fill in the following blank:

```
scm> __________________
(a 3 c)
```

```
`(a ,(a b c) c)
```

Additional Practice Now, let the following be defined:

```
scm> (define condition '(= 1 1))
condition
scm> (define if-true '(print 3))
if-true
scm> (define if-false '(print 5))
if-false
```

Write the Scheme expression **using the variables condition, if-true, and if-false** along with quasiquoting, that would evaluate to the following:

```
scm> ________________________
(if (= 1 1) (print 3) (print 5))
```

```
`(if ,condition ,if-true ,if-false)
```

Note: This worksheet is a problem bank—most TAs will not cover all the problems in discussion section.

If Program Scheme

Q3: If Program Scheme

Now let's try writing the `if_program` Python example using programs as data in Scheme! There are quite a few similarities between Python and Scheme, but we have to make a few adjustments when converting our code over to Scheme. We'll start out by writing a Scheme function with the `define` form we use for normal functions.

First, let's consider the following questions:

In the Python `if_program`, we returned a **string** that, when evaluated, would execute an `if` statement with the correct parameters. In Scheme, we won't return a string, but we'll return something else that can represent an unevaluated expression. What type will we return for Scheme? Here, what "type" refers to the data type: function, list, integer, string, etc..

Scheme List

In the Python `if_program`, our operands were all **strings** representing the `condition`, return value if `true`, and return value if `false`. Expressing our operands as strings allowed them to remain unevaluated lines of code. What will each of our operands be in Scheme (i.e. how do we pass in the unevaluated versions of our operand expressions in Scheme)? Give an example of an acceptable operand expression for the `condition` parameter.

Our operands will be the *quoted* versions of each of our desired operands. I.e. for condition, we can pass in `'(= 1 1)`. Note that the operand `'(= 1 1)` will still be evaluated by the Scheme interpreter before being passed into the function (since we're still following normal call expression rules at this point), but the result of evaluating `'(= 1 1)` is the list `(= 1 1)`. So we end up passing in the list `(= 1 1)` into our function instead of `#t`.

Let's write this out!

Write a function `if-function` using the `define` form, which will take in the following parameters:

1. `condition` : a quoted expression which will evaluate to the condition in our if expression
2. `if-true` : a quoted expression which will evaluate to the value we want to return if true (`#t`)
3. `if-false` : a quoted expression which will evaluate to the value we want to return if false (`#f`)

and returns a Scheme list representing the expression that, when evaluated, will evaluate to the result of our `if` expression.

Here are some doctests to show this:

```
scm> (if-program '(= 0 0) '2 '3)
(if (= 0 0) 2 3)
scm> (eval (if-program '(= 0 0) '2 '3))
2
scm> (if-program '(= 1 0) '(print 3) '(print 5))
(if (= 1 0) (print 3) (print 5))
scm> (eval (if-program '(= 1 0) '(print 3) '(print 5)))
5
```

> **Hint:** You may want to look at your solution to the Additional Practice subproblem of "Writing Quasiquote Expressions"!

```
(define (if-program condition if-true if-false)
  `(if ,condition ,if-true ,if-false)
)
```

Note: We already have a built-in `if` special form in Scheme that behaves like `if-program`, so this example isn't as exciting, but with this new strategy of treating programs as data we can think about starting to define our *own* special forms that aren't built into Scheme!

Programs As Data Practice

But before we start worrying too much about defining custom special forms, let's practice more with writing procedures that take in, manipulate, and output Scheme expressions as lists!

While doing these problems, remember that quasiquotation is not the only way to create Scheme lists! We also have `list` and `cons`. The optimal approach for creating the desired expression as a list depends on the problem.

Q4: Exponential Powers

Implement the procedure `pow-expr`, which takes in a base `n` and a nonnegative integer `p`. The procedure should create a program as a list that, when passed into the `eval` procedure, evaluates the value `n^p`, or `n` to the `p`th power.

For example, the expression `(pow-expr 2 5)` returns a program as a list. When `eval` is called on that returned list, it should evaluate to `2^5`, which is 32. We'll do this by building nested multiplication expressions!

```
scm> (define expr (pow-expr 5 1))
scm> expr
(* 1 5)
scm> (eval expr)
5

scm> (define expr2 (pow-expr 5 2))
scm> expr2
(* (* 1 5) 5)
scm> (eval expr2)
25
```

Hint: Note that the expression returned by `(pow-expr 5 2)` is just the expression returned by `(pow-expr 5 1)` nested in another multiplication expression. There is an inherent *recursive* structure here, so what programming paradigm do you think we should use?

```
(define (pow-expr n p)
    (if (= p 0) 1
        `(* ,(pow-expr n (- p 1)) ,n)))
```

Alternate solution using `list`:

```
(define (pow-expr n p)
    (if (= p 0) 1
        (list '* (pow-expr n (- p 1)) n)))
```

Q5: Swap

Implement `swap`, a procedure which takes a call expression `expr` and returns the same expression with its first two operands swapped if the value of the second operand is greater than the value of the first. Otherwise, it should just return the original expression.

For example, `(swap '(- 1 (+ 3 5) 7))` should return the expression `(- (+ 3 5) 1 7)` since `1` evaluates to `1`, `(+ 3 5)` evaluates to `8`, and `8 > 1`. Any operands after the first two should not be evaluated during the execution of the procedure, and they should be left unchanged in the final expression. You may assume that every operand evaluates to a number and that there are always at least two operands in `expr`.

```
scm> (swap '(- 1 (+ 3 5) 7 9))
(- (+ 3 5) 1 7 9)

scm> (swap '(* (+ 1 1) (- 2 1)))
(* (+ 1 1) (- 2 1))
```

We have provided a `let` expression in addition to the provided procedures to help simplify your code.

```
(define (cddr s)
  (cdr (cdr s))
)

(define (cadr s)
  (car (cdr s))
)

(define (caddr s)
  (car (cddr s))
)

(define (swap expr)
    (let ((op (car expr))
        (first (car (cdr expr)))
        (second (caddr expr))
        (rest (cdr (cddr expr))))
        (if (> (eval second) (eval first))
            (cons op (cons second (cons first rest)))
            expr)
    )
)
```

Q6: Make Or

Implement make-or, which returns, as a list, a program that takes in two expressions and or's them together (applying short-circuiting rules). However, do this without using the or special form. You may also assume the name v1 doesn't appear anywhere outside this function. For a quick reminder on the short-circuiting rules for or take a look at slide 18 of Lecture 3 on Control.

The behavior of the or procedure is specified by the following doctests:

```
scm> (define or-program (make-or '(print 'bork) '(/ 1 0)))
or-program
scm> (eval or-program)
bork
scm> (eval (make-or '(= 1 0) '(+ 1 2)))
3
```

As a starting point, consider this ***incorrect*** implementation:

```
scm> (define (make-or expr1 expr2) `(if ,expr1 ,expr1 ,expr2)))
```

This incorrect implementation would result in the following incorrect behavior:

```
scm> (make-or '(print 'bork) '(/ 1 0))
(if (print (quote bork)) (print (quote bork)) (/ 1 0))
scm> (eval (make-or '(print 'bork) '(/ 1 0)))
bork
bork
```

Although there is short-circuiting behavior, notice that (print (quote bork)) is evaluated twice, once when evaluating the <predicate> expression and another time when evaluating the <if-true> expression. Consider how we can get around this using the let special form.

```
(define (make-or expr1 expr2)
    `(let ((v1 ,expr1))
        (if v1 v1 ,expr2))
)
```

Q7: Make "Make Or"

The above code generates a program that evaluates an `or` expression without using any `or` statements. However, we can take it even one step further: let's create a program which generates `make-or`, the program you created which generates an `or` expression.

Implement `make-make-or`, a program which generates a program which, when `eval`'d, can be `apply`'d to make an `or` expression with differing variables. You may find the code you wrote above to be useful.

> Hint: recall that you want to construct a list that resembles the program. Do you know what this list would look like?

```
(define (make-make-or)
  '(define (make-or expr1 expr2) `(let ((v1 ,expr1)) (if v1 v1 ,expr2)))
)
```

Now, given this function, determine the outputs from the following expressions:

```
scm> (make-make-or)
```

(define (make-or expr1 expr2) (quasiquote (let ((v1 (unquote expr1))) (if v1 v1 (unquote expr2)))))

```
scm> (eval (make-make-or))
```

make-or

```
scm> (eval (eval (make-make-or)))
```

(lambda (expr1 expr2) (quasiquote (let ((v1 (unquote expr1))) (if v1 v1 (unquote expr2)))))

```
scm> (apply (eval (eval (make-make-or))) '(#t (/ 1 0)))
```

(let ((v1 #t)) (if v1 v1 (/ 1 0)))

```
scm> (eval (apply (eval (eval (make-make-or))) '(#t (/ 1 0))))
```

#t