CS 61A
Summer 2018

Interpreters and Macros

Discussion 11: July 26, 2018 Solutions

1 Calculator

An interpreter is a program that understands other programs. Today, we will explore how to interpret a simple language that uses Scheme syntax called *Calculator*.

The Calculator language includes only the four basic arithmetic operations: $+$, $-$, $*$, and $/$. These operations can be nested and can take any numbers of arguments. A few examples of calculator expressions and their corresponding values are given on the right.

```
calc> (+ 2 2)
4

calc> (- 5)
-5

calc> (* (+ 1 2) (+ 2 3))
15
```

Reading expressions

Recall that the reader component of an interpreter parses input strings and represents them as data structures in the implementing language. In this case, we need to represent Calculator expressions as Python objects. To represent numbers, we can just use Python numbers. To represent the names of the arithmetic procedures, we can use Python strings (e.g. `'+'`).

Call expressions are a bit more complicated. First, note that like Scheme call expressions, call expressions in Calculator look just like Scheme lists. For example, to construct the expression `(+ 2 3)` in Scheme, we would do the following:

```
scm> (cons '+ (cons 2 (cons 3 nil)))
(+ 2 3)
```

To represent Scheme lists in Python, we will use the `Pair` class. A `Pair` instance holds exactly two elements. Accordingly, the `Pair` constructor takes in two arguments, and to make a list we must nest calls to the constructor and pass in `nil` as the second element of the last pair. Note that in our implementation, `nil` is bound to a special user-defined object that represents an empty list, whereas `nil` in Scheme is actually an empty list.

```
>>> Pair('+', Pair(2, Pair(3, nil)))
Pair('+', Pair(2, Pair(3, nil)))
```

Each `Pair` instance has two instance attributes: `first` and `second`, which are bound to the first and second elements of the pair respectively.

```
>>> p = Pair('+', Pair(2, Pair(3, nil)))
>>> p.first
'+'
>>> p.second
Pair(2, Pair(3, nil))
>>> p.second.first
2
```

Here's an implementation of what we described:

```
class Pair:
    """Represents the built-in pair data structure in Scheme."""
    def __init__(self, first, second):
        self.first = first
        self.second = second

    def map(self, fn):
        """Maps fn to every element in a well-formed list, returning a new
        Pair.

        >>> Pair(1, Pair(2, Pair(3, nil))).map(lambda x: x * x)
        Pair(1, Pair(4, Pair(9, nil)))
        """
        assert isinstance(self.second, Pair) or self.second is nil, \
            "Second element in pair must be another pair or nil"
        return Pair(fn(self.first), self.second.map(fn))

    def __getitem__(self, i):
        """Allows us to index into well-formed lists and treat well-formed
        lists like Python iterables.

        >>> p = Pair(1, Pair(2, Pair(3, nil)))
        >>> p[1]
        2
        >>> list(p)
        [1, 2, 3]
        """
        assert isinstance(self.second, Pair) or self.second is nil, \
            "Second element in pair must be another pair or nil"
        if i == 0:
            return self.first
        return self.second[i - 1]

    def __repr__(self):
        return 'Pair({}, {})'.format(self.first, self.second)

class nil:
    """Represents the special empty pair nil in Scheme."""
    def map(self, fn):
        return nil
    def __getitem__(self, i):
        raise IndexError('Index out of range')
    def __repr__(self):
        return 'nil'

nil = nil() # this hides the nil class *forever*
```

Questions

1.1 Write out constructor calls to create `Pair` objects representing the following Calculator expression.

```
> (/ 1 2 3)
```

```
>>> Pair('/', Pair(1, Pair(2, Pair(3, nil))))
```

```
> (+ 1 2 (- 3 4))
```

```
>>> Pair('+', Pair(1, Pair(2, Pair(
        Pair('-', Pair(3, Pair(4, nil))), nil))))
```

```
> (+ 1 (* 2 3) 4)
```

```
>>> Pair('+', Pair(1, Pair(Pair( '*', Pair(2, Pair(3, nil))),
        Pair(4,nil))))
```

Video walkthrough

1.2 Write out the Calculator expression with proper syntax that corresponds to the following `Pair` constructor calls.

```
>>> Pair('+', Pair(1, Pair(2, Pair(3, Pair(4, nil)))))
```

```
> (+ 1 2 3 4)
```

```
>>> Pair('+', Pair(1, Pair(Pair('*', Pair(2, Pair(3, nil))), nil)))
```

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

Video walkthrough

1.3 Answer the following questions about a `Pair` instance representing the Calculator expression `(+ 2 4 6 8)`.

i. Write out the Python expression that returns a `Pair` representing the given expression.

```
>>> Pair('+', Pair(2, Pair(4, Pair(6, Pair(8, nil)))))
```

ii. What is the operator of the call expression? Given that the `Pair` you constructed in the previous part was bound to the name `p`, how would you retrieve the operator?

`p.first`

iii. What are the operands of the call expression? Given that the `Pair` you constructed in Part (i) was bound to the name `p`, how would you retrieve a list containing all of the operands? How would you retrieve only the first operand?

`p.second` to get a list containing all the operands. `p.second.first` to get the first operand by itself.

Evaluation

The evaluation component of an interpreter determines the type of an expression and executes corresponding evaluation rules.

Here are the evaluation rules for the three types of Calculator expressions:

1. **Numbers** are self-evaluating. For example, the numbers 3.14 and 165 just evaluate to themselves.
2. **Names** are looked up in the `OPERATORS` dictionary. Each name (e.g. `'+'`) is bound to a corresponding function in Python that does the appropriate operation on a list of numbers (e.g. `sum`).
3. **Call expressions** are evaluated the same way you've been doing them all semester:

   (1) **Evaluate** the operator, which evaluates to a function.

   (2) **Evaluate** the operands from left to right.

   (3) **Apply** the function to the value of the operands.

The function `calc_eval` takes in a Calculator expression represented in Python and implements each of these rules:

```
def calc_eval(exp):
    """Evaluates a Calculator expression represented as a Pair."""
    if isinstance(exp, Pair): # Call expressions
        fn = calc_eval(exp.first)
        args = list(exp.second.map(calc_eval))
        return calc_apply(fn, args)
    elif exp in OPERATORS:    # Names
        return OPERATORS[exp]
    else:                     # Numbers
        return exp
```

Note that `calc_eval` is recursive! In order to evaluate call expressions, we must call `calc_eval` on the operator and each of the operands.

The *apply* step in the Calculator language is straight-forward, since we only have primitive procedures. This step is more complex when it comes to applying Scheme procedures, which may include user-defined procedures.

Given the Python function that implements the appropriate Calculator operation and a Python list of numbers, the `calc_apply` function simply calls the function on the arguments, and regular Python evalutation rules take place.

```
def calc_apply(fn, args):
    """Applies a Calculator operation to a list of numbers."""
    return fn(args)
```

Questions

1.1 How many calls to `calc_eval` and `calc_apply` would it take to evaluate each of the following Calculator expressions?

```
> (+ 2 4 6 8)
```

6 calls to eval: 1 for the entire expression, and then 1 each for the operator and each operand.

1 call to apply the addition operator.

```
> (+ 2 (* 4 (- 6 8)))
```

10 calls to eval: 1 for the whole expression, then 1 for each of the operators and operands. When we encounter another call expression, we have to evaluate the operators and operands inside as well.

3 calls to apply the function the arguments for each call expression.
Video walkthrough

1.2 Suppose we want to add handling for comparison operators `>`, `<`, and `=` as well as `and` expressions to our Calculator interpreter. These should work the same way they do in Scheme.

```
calc> (and (= 1 1) 3)
3
calc> (and (+ 1 0) (< 1 0) (/ 1 0))
#f
```

i. Are we able to handle expressions containing the comparison operators with the existing implementation of `calc_eval`? Why or why not?

Comparison expressions are regular call expressions, so we need to evaluate the operator and operands and then apply a function to the arguments. Therefore, we do not need to change `calc_eval`. We simply need to add new entries to the `OPERATORS` dictionary that map `'<'`, `'>'`, and `'='` to functions that perform the appropriate comparison operation.

ii. Are we able to handle `and` expressions with the existing implementation of `calc_eval`? Why or why not?

Since `and` is a special form that short circuits on the first false-y operand, we cannot handle these expressions the same way we handle call expressions. We need to add special handling for combinations that don't evaluate all the operands.

iii. Now, complete the implementation below to handle `and` expressions. You may assume the conditional operators (e.g. `<`, `>`, `=`, etc) have already been implemented for you.

```
def calc_eval(exp):
    if isinstance(exp, Pair):
        if _________________________: # and expressions
```

```
            return eval_and(exp.second)
        else:                       # Call expressions
            return calc_apply(calc_eval(exp.first), list(exp.second.map(calc_eval)))
    elif exp in OPERATORS:          # Names
        return OPERATORS[exp]
    else:                           # Numbers
        return exp

def eval_and(operands):


def calc_eval(exp):
    if isinstance(exp, Pair):
        if exp.first == 'and':  # and expressions
            return eval_and(exp.second)
        else:                   # Call expressions
            return calc_apply(calc_eval(exp.first), list(exp.second.map(calc_eval)))
    elif exp in OPERATORS:      # Names
        return OPERATORS[exp]
    else:                       # Numbers
        return exp

def eval_and(operands):
    curr, val = operands, True
    while curr is not nil:
        val = calc_eval(curr.first)
        if val is False:
            return False
        curr = curr.second
    return val
```

2 Macros

So far we've been able to define our own procedures in Scheme using the `define` special form. When we call these procedures, we have to follow the rules for evaluating call expressions, which involve evaluating all the operands.

We know that special form expressions do not follow the evaluation rules of call expressions. Instead, each special form has its own rules of evaluation, which may include not evaluating all the operands. Wouldn't it be cool if we could define our own special forms where we decide which operands are evaluated? Consider the following example where we attempt to write a function that evaluates a given expression twice:

```
scm> (define (twice f) (begin f f))
twice
scm> (twice (print 'woof))
woof
```

Since `twice` is a regular procedure, a call to `twice` will follow the same rules of evaluation as regular call expressions; first we evaluate the operator and then we evaluate the operands. That means that `woof` was printed when we evaluated the operand `(print 'woof)`. Inside the body of `twice`, the name `f` is bound to the value `undefined`, so the expression `(begin f f)` does nothing at all!

The problem here is clear: we need to prevent the given expression from evaluating until we're inside the body of the procedure. This is where the `define-macro` special form, which has identical syntax to the regular `define` form, comes in:

```
scm> (define-macro (twice f) (list 'begin f f))
twice
```

`define-macro` allows us to define what's known as a **macro**, which is simply a way for us to combine unevaluated input expressions together into another expression. When we call macros, the operands are not evaluated, but rather are treated as Scheme data. This means that any operands that are call expressions or special form expression are treated like lists.

If we call `(twice (print 'woof))`, `f` will actually be bound to the list `(print 'woof)` instead of the value `undefined`. Inside the body of `define-macro`, we can insert these expressions into a larger Scheme expression. In our case, we would want a `begin` expression that looks like the following:

```
(begin (print 'woof) (print 'woof))
```

As Scheme data, this expression is really just a list containing three elements: `begin` and `(print 'woof)` twice, which is exactly what `(list 'begin f f)` returns. Now, when we call `twice`, this list is evaluated as an expression and `(print 'woof)` is evaluated twice.

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

Quasiquoting

Recall that the `quote` special form prevents the Scheme interpreter from executing a following expression. We saw that this helps us create complex lists more easily than repeatedly calling `cons` or trying to get the structure right with `list`. It seems like this form would come in handy if we are trying to construct complex Scheme expressions with many nested lists.

```
scm> (define a 1)
a
scm> '(cons a nil)
(cons a nil)
```

Consider that we rewrite the `twice` macro as follows:

```
(define-macro (twice f)
  '(begin f f))
```

This seems like it would have the same effect, but since the `quote` form prevents any evaluation, the resulting expression we create would actually be `(begin f f)`, which is not what we want.

The **quasiquote** allows us to construct literal lists in a similar way as quote, but also lets us specify if anything within the operand should be evaluated.

At first glance, the quasiquote (which can be invoked with the backtick `` ` `` or the `quasiquote` special form) behaves exactly the same as `'` or `quote`. However, using quasiquotes gives you the ability to **unquote** (which can be invoked with the comma `,` or the `unquote` special form). This removes an expression from the quoted context, evaluates it, and places it back in.

```
scm> `(cons a nil)
(cons a nil)
scm> `(cons ,a nil)
(cons 1 nil)
```

By combining quasiquotes and unquoting, we can often save ourselves a lot of trouble when building macro expressions.

Here is how we could use quasiquoting to rewrite our previous example:

```
(define-macro (twice f)
  `(begin ,f ,f))
```

Questions

2.1 Write a macro that takes an expression and returns a parameter-less lamba function with the expression as its body

```
(define-macro (make-lambda expr)
```

With quasiquotes:

```
(define-macro (make-lambda expr)
    `(lambda () ,expr))
```

With the list constructor:

```
(define-macro (make-lambda expr)
      (list 'lambda (list) expr))
```

```
scm> (make-lambda (print 'hi))
(lambda () (print (quote hi)))
scm> (make-lambda (/ 1 0))
```

```
(lambda () (/ 1 0))
scm> (define print-3 (make-lambda (print 3)))
print-3
scm> (print-3)
3
```

2.2 Write a macro 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 of our macro. Fill in the implementation below.

```
(define-macro (or-macro expr1 expr2)

  `(let ((v1 ____________________________________________________________________________))

     (if  ____________________________________________________________________________

       ____________________________________________________________________________)))
```

```
scm> (or-macro (print 'bork) (/ 1 0))
bork
scm> (or-macro (= 1 0) (+ 1 2))
3
```

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

2.3 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.

(a) Fill in the skeleton below to implement this without using quasiquotes.

```
(define-macro (when condition . exprs)

  (list 'if_____________________________________________________________________________))
```

```
(define-macro (when condition . exprs)
      (list 'if condition (cons 'begin exprs) ''okay))
```

(b) Now, implement the macro using quasiquotes.

```
(define-macro (when condition . exprs)

  `(if _______________________________________________________________________________))
```

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

Extra questions

2.4 Write a macro that takes an expression and a number `n` and repeats the expression `n` times. For example, `(repeat-n expr 2)` should behave the same as `(twice expr)`. Note that it's possible to pass in a combination as the second argument (e.g. `(+ 1 2)`) as long as it evaluates to a number. Be sure that you evaluate this expression in your macro so that you don't treat it as a list.

Complete the implementation below, making use of the `replicate` function from Discussion 7.

```
(define (replicate x n)
  (if (= n 0) nil
    (cons x (replicate x (- n 1)))))

(define-macro (repeat-n expr n)
```

```
(define-macro (repeat-n expr n)
    (cons 'begin (replicate expr (eval n))))
```

```
scm> (repeat-n (print '(resistance is futile)) 3)
(resistance is futile)
(resistance is futile)
(resistance is futile)
scm> (repeat-n (print (+ 3 3)) (+ 1 1))  ; Pass a call expression in as n
6
6
```

2.5 Write a macro that takes in a call expression and strips out every other argument. The first argument is kept, the second is removed, and so on. You may find it helpful to write a helper function.

```
(define-macro (prune-expr expr)


(define (prune lst)
  (if (or (null? lst) (null? (cdr lst))) lst
      (cons (car lst) (prune (cdr (cdr lst))))))

(define-macro (prune-expr expr)
    (cons (car expr) (prune (cdr expr))))

scm> (prune-expr (+ 10))
10
scm> (prune-expr (+ 10 100))
10
scm> (prune-expr (+ 10 100 1000))
1010
scm> (prune-expr (prune-expr (+ 10 100) 'garbage))
10
```