CS 61A
Fall 2019

Macros and Streams

Discussion 11: November 13, 2019

1 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
```

To recap, macros are called similarly to regular procedures, but the rules for evaluating them are different. We evaluated lambda procedures in the following way:

1. Evaluate operator
2. Evaluate operands
3. Apply operator to operands, evaluating the body of the procedure

However, the rules for evaluating calls to macro procedures are:

1. Evaluate operator
2. Apply operator to unevaluated operands
3. Evaluate the expression returned by the macro in the frame it was called in.

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 any sub-expression within the list 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

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

```
(define-macro (make-lambda 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
```

1.2 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 given below. The `replicate` function takes in a value `x` and a number `n` and returns a list with `x` repeated `n` times.

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

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

1.3 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
```

2 Streams

In Python, we can use iterators to represent infinite sequences (for example, the generator for all natural numbers). However, Scheme does not support iterators. Let's see what happens when we try to use a Scheme list to represent an infinite sequence of natural numbers:

```
scm> (define (naturals n)
         (cons n (naturals (+ n 1))))
naturals
scm> (naturals 0)
Error: maximum recursion depth exceeded
```

Because `cons` is a regular procedure and both its operands must be evaluted before the pair is constructed, we cannot create an infinite sequence of integers using a Scheme list. Instead, our Scheme interpreter supports *streams*, which are *lazy* Scheme lists. The first element is represented explicitly, but the rest of the stream's elements are computed only when needed. Computing a value only when it's needed is also known as *lazy evaluation*.

```
scm> (define (naturals n)
         (cons-stream n (naturals (+ n 1))))
naturals
scm> (define nat (naturals 0))
nat
scm> (car nat)
0
scm> (cdr nat)
#[promise (not forced)]
scm> (car (cdr-stream nat))
1
scm> (car (cdr-stream (cdr-stream nat)))
2
```

We use the special form `cons-stream` to create a stream:

```
(cons-stream <operand1> <operand2>)
```

`cons-stream` is a special form because the second operand is not evaluated when evaluating the expression. To evaluate this expression, Scheme does the following:

1. Evaluate the first operand.
2. Construct a promise containing the second operand.
3. Return a pair containing the value of the first operand and the promise.

To actually get the rest of the stream, we must call `cdr-stream` on it to force the promise to be evaluated. Note that this argument is only evaluated once and is then stored in the promise; subsequent calls to `cdr-stream` returns the value without recomputing it. This allows us to efficiently work with infinite streams like the `naturals` example above. We can see this in action by using a non-pure function to compute the rest of the stream:

```
scm> (define (compute-rest n)
...>   (print 'evaluating!)
...>   (cons-stream n nil))
compute-rest
scm> (define s (cons-stream 0 (compute-rest 1)))
s
scm> (car (cdr-stream s))
evaluating!
1
scm> (car (cdr-stream s))
1
```

Here, the expression `compute-rest 1` is only evaluated the first time `cons-stream` is called, so the symbol `evaluating!` is only printed the first time.

When displaying a stream, the first element of the stream and the promise are displayed separated by a dot (this indicates that they are part of the same pair, with the promise as the `cdr`). If the value in the promise has not been evaluated by calling `cdr-stream`, we consider it to be not forced. Otherwise, we consider it forced.

```
scm> (define s (cons-stream 1 nil))
s
scm> s
(1 . #[promise (not forced)])
scm> (cdr-stream s) ; nil
()
scm> s
(1 . #[promise (forced)])
```

Streams are very similar to Scheme lists in that they are also recursive structures. Just like the `cdr` of a Scheme list is either another Scheme list or `nil`, the `cdr-stream` of a stream is either a stream or `nil`. The difference is that whereas both arguments to `cons` are evaluated upon calling `cons`, the second argument to `cons-stream` isn't evaluated until the first time that `cdr-stream` is called.

Here's a summary of what we just went over:

- `nil` is the empty stream
- `cons-stream` constructs a stream containing the value of the first operand and a promise to evaluate the second operand
- `car` returns the first element of the stream
- `cdr-stream` computes and returns the rest of stream

Questions

2.1 What would Scheme display?

As you work through these problems, remember that streams have two important components:

- Lazy evaluation – so the remainder of the stream isn't computed until explicitly requested.
- Memoization – so anything we compute won't be recomputed.

The examples here stretch these concepts to the limit. In most practical use cases, you may find you rarely need to redefine functions that compute the remainder of the stream.

```
scm> (define (has-even? s)
       (cond ((null? s) #f)
             ((even? (car s)) #t)
             (else (has-even? (cdr-stream s)))))
has-even?
scm> (define (f x) (* 3 x))
f
scm> (define nums (cons-stream 1 (cons-stream (f 3) (cons-stream (f 5) nil))))
nums
scm> nums

scm> (cdr nums)

scm> (cdr-stream nums)

scm> nums

scm> (define (f x) (* 2 x))
f
scm> (cdr-stream nums)

scm> (cdr-stream (cdr-stream nums))

scm> (has-even? nums)
```

2.2 Using streams can be tricky! Compare the following two implementations of `filter-stream`, the first is a correct implementation whereas the second is wrong in some way. What's wrong with the second implementation?

```
; Correct
(define (filter-stream f s)
  (cond
    ((null? s) nil)
    ((f (car s)) (cons-stream (car s) (filter-stream f (cdr-stream s))))
    (else (filter-stream f (cdr-stream s)))))

; Incorrect
(define (filter-stream f s)
  (if (null? s) nil
    (let ((rest (filter-stream f (cdr-stream s))))
      (if (f (car s))
        (cons-stream (car s) rest)
        rest))))
```

2.3 Write a function `map-stream`, which takes a function `f` and a stream `s`. It returns a new stream which has all the elements from `s`, but with `f` applied to each one.

```
(define (map-stream f s)
```

```
scm> (define evens (map-stream (lambda (x) (* x 2)) nat))
evens
scm> (car (cdr-stream evens))
2
```

2.4 Write a function `slice` which takes in a stream `s`, a `start`, and an `end`. It should return a Scheme list that contains the elements of `s` between index `start` and `end`, not including `end`. If the stream ends before `end`, you can return `nil`.

```
(define (slice s start end)
```

```
scm> (define nat (naturals 0)) ; See naturals procedure defined earlier
nat
scm> (slice nat 4 12)
(4 5 6 7 8 9 10 11)
```

2.5 Since streams only evaluate the next element when they are needed, we can combine infinite streams together for interesting results! Use it to define a few of our favorite sequences. We've defined the function `combine-with` for you below, as well as an example of how to use it to define the stream of even numbers.

```
(define (combine-with f xs ys)
  (if (or (null? xs) (null? ys))
    nil
    (cons-stream
      (f (car xs) (car ys))
      (combine-with f (cdr-stream xs) (cdr-stream ys)))))
scm> (define evens (combine-with + (naturals 0) (naturals 0)))
evens
scm> (slice evens 0 10)
(0 2 4 6 8 10 12 14 16 18)
```

For these questions, you may use the `naturals` stream in addition to `combine-with`.

i. `(define factorials`

```
scm> (slice factorials 0 10)
(1 1 2 6 24 120 720 5040 40320 362880)
```

ii. `(define fibs`

```
scm> (slice fibs 0 10)
(0 1 1 2 3 5 8 13 21 34)
```

iii. (Extra for practice) Write `exp`, which returns a stream where the nth term represents the degree-n polynomial expantion for e^x, which is $\sum_{i=0}^{n} x^i/i!$.

You may use `factorials` in addition to `combine-with` and `naturals` in your solution.

`(define (exp x)`

```
scm> (slice (exp 2) 0 5)
(1 3 5 6.333333333 7)
```

Extra Questions

2.1 Using the `make-lambda` macro you defined Question 1, define `make-stream`, a macro which returns a pair of elements, where the second element is not evaluated until `cdr-stream` is called on it. Also define the procedure `cdr-stream`, which takes in a stream returned by `make-stream` and returns the result of evaluating the second element in the stream pair.

Unlike the streams we've seen in lecture and earlier in discussion, if you repeatedly call `cdr-stream` on a stream returned by `make-stream`, you may evaluate an expression multiple times.

```
(define-macro (make-stream first second)




(define (cdr-stream stream)
```

```
scm> (define a (make-stream (print 1) (make-stream (print 2) nil)))
1
a
scm> (define b (cdr-stream a))
2
b
scm> (cdr-stream b)
()
```

2.2 We can even represent the sequence of all prime numbers as an infinite stream! Define a function `sieve`, which takes in a stream of increasing numbers and returns a stream containing only those numbers which are not multiples of an earlier number in the stream. We can define `primes` by sifting all natural numbers starting at 2. Look online for the **Sieve of Eratosthenes** if you need some inspiration.
Hint: You might find using `filter-stream` as defined earlier helpful.

```
(define (sieve s)
```

```
(define primes
  (sieve (naturals 2)))
scm> (slice primes 0 10)
(2 3 5 7 11 13 17 19 23 29)
```