CS 61A Spring 2018 — Macros & Streams

Discussion 9: April 11, 2018 Solutions

1 Macros

So far, we've mostly explored similarities between the Python and Scheme languages. For example, the Scheme list data structure is a close analogue to the Python linked list. As another example, we saw how tail-call optimization allows us to write recursive Scheme functions that use a constant amount of space. This makes it feasible to translate iterative code from Python.

On the other hand, **macros** are a Scheme feature that don't have a apparent Python equivalent. Like functions, macros are a useful tool for simplifying code via abstraction. But while functions typically operate on values like numbers and lists, macros have the option of transforming unevaluated code, leading to a whole new world of possibilities!

As a reminder, most Scheme functions do not have side effects. One exception to this is `print`. Just like in Python, `print` doesn't return anything! With that in mind, let's consider an example where we want to repeat a piece of code twice.

```
scm> (print 1)
1
scm> (print 'hello)
hello
scm> (print '(yes this is dog))
(yes this is dog)
```

```
(print 'woof)
```

A first attempt at this might be:

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

Remember that `print` doesn't return anything! So we would only see the first call to `print` in this case. The problem here is clear: we need to prevent the expression we want to double from evaluating, and then somehow call it twice.

As an example of this, imagine if the problem were less constrained and we could surround our original expression in a `define` expression. In that case, we could use higher order functions to get what we want:

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

But if the expression is given to us directly, there's no way to "undo" the execution and delay it for later!

```
scm> (define (twice result)
       (begin
         (define (f) result) % This won't work!
         (f)(f)))
twice
scm> (twice (print 'woof))
woof
```

Clearly, we need a special form, since we cannot evaluate our operand immediately. This is where we apply the `define-macro` special form.

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

This looks a bit like a function definition. `twice` is the name of the macro, and everything that follows in the same list is a required parameter. When we evaluate the macro form, we won't evaluate any parameters immediately. Instead, the body of the macro describes the final expression we want to evaluate, with the unevaluated parameters put in place! Recall that we want a final expression that looks like:

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

Now, let `f` be the snippet of `print` code from earlier (not the result of evaluation, which is simply nothing) The expression:

```
(list 'begin f f)
```

Creates our desired expression, and then finally evaluates it. Note that if we used:

```
'(begin f f)
```

This wouldn't work, since `f` would stay as `f` and wouldn't be replaced with our `print` expression. However, this seems easier to do than calling `list` a bunch of times. Is there a way to get the best of both worlds?

1.1 Quasiquoting

Recall that the `quote` special form prevents the Scheme interpreter from executing a following expression. You may have used it in the past to create lists without needing to call functions such as `cons` and `list`. However, you cannot create any lists that depend on the results of function evaluation due to the fact that quoting will surpress all evaluation. This is not the case with quasiquoting.

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

At first glance, the **quasiquote** (which can be invoked with the backtick ` or the `quasiquote` special form) behaves exactly the same.

However, using quasiquotes gives you the ability to **unquote** (which can be invoked with the 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 1 nil)
```

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

As one last example, we can create a quasiquoted version of our macro from earlier:

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

Questions

1.1 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)`.

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

```
(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)) 4)
(resistance is futile)
(resistance is futile)
(resistance is futile)
(resistance is futile)
```

1.2 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)))
```

Note that the printed version of this discussion in Spring 2018 was buggy – it was missing the quasiquote in the skeleton.

1.3 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
```

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 the second argument to `cons` is always evaluated, 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> (car (cdr-stream nat))
1
scm> (car (cdr-stream (cdr-stream nat)))
2
```

We use the special form `cons-stream` to create a stream. Note that `cons-stream` is a special form, because the second operand `(naturals (+ n 1)))` is *not* evaluated when `cons-stream` is called. It's only evaluated when `cdr-stream` is used to inspect the rest of the stream.

- `nil` is the empty stream
- `cons-stream` creates a non-empty stream from an initial element and an expression to compute the rest of the stream
- `car` returns the first element of the stream
- `cdr-stream` computes and returns the rest of stream

Streams are very similar to Scheme lists. The `cdr` of a Scheme list is either another Scheme list or `nil`; likewise, the `cdr-stream` of a stream is either a stream or `nil`. The difference is that the expression for the rest of the stream is computed the first time that `cdr-stream` is called, instead of when `cons-stream` is used. Subsequent calls to `cdr-stream` return this 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
```

Note that the symbol `evaluating!` is only printed the first time `cdr-stream` is called.

Video walkthrough

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
```

```
(1 . #[promise (not forced)])
```

```
scm> (cdr nums)
```

```
#[promise (not forced)]
```

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

```
(9 . #[promise (not forced)])
```

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

```
(9 . #[promise (not forced)])
```

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

```
True
```

Video walkthrough

2.2 Write a function `range-stream` which takes a `start` and `end`, and returns a stream that represents the integers between `start` and `end - 1` (inclusive).

```
(define (range-stream start end)

  (if (________________________________________________________________________________)

    nil

    (cons-stream ______________________________________________________________________)))
```

```
  (if (= start end)
    nil
    (cons-stream start (range-stream (+ start 1) end))))
```

It might help to compare this to the version of `range` for regular (non-stream) Scheme lists:

```
(define (range start end)
  (if (= start end)
    nil
    (cons start (range (+ start 1) end))))
```

```
scm> (define s (range-stream 1 5))
s
scm> (car (cdr-stream s))
2
```

2.3 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)

  (cond
    ((or (null? s) (= end 0)) nil)
    ((> start 0)
      (slice (cdr-stream s) (- start 1) (- end 1)))
    (else
      (cons (car s)
        (slice (cdr-stream s) (- start 1) (- end 1))))))
```

```
scm> (slice nat 4 12)
(4 5 6 7 8 9 10 11)
```

2.4 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`.

(Continued on the next page)

i. `(define factorials`

```
  (cons-stream 1 (combine-with * (naturals 1) factorials)))
```

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

ii. `(define fibs`

```
  (cons-stream 0
    (cons-stream 1
      (combine-with + fibs (cdr-stream fibs)))))
```

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

iii. 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)
  (let ((terms (combine-with (lambda (a b) (/ (expt x a) b))
                             (cdr-stream (naturals 0))
                             (cdr-stream factorials))))
    (cons-stream 1 (combine-with + terms (exp x)))))
scm> (slice (exp 2) 0 5)
(1 3 5 6.333333333 7 7.266666667)
```