CS 61A Fall 2022 — Macros, Tail Calls

Discussion 12: November 16, 2022

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:

- Evaluate operator
- Evaluate operands
- Apply operator to operands, evaluating the body of the procedure

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

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

Q1: Shapeshifting Macros

When writing macros in Scheme, we often create a list of symbols that evaluates to a desired Scheme expression. In this question, we'll practice different methods of creating such Scheme lists.

We have executed the following code to define `x` and `y` in our current environment.

```
(define x '(+ 1 1))
(define y '(+ 2 3))
```

We want to use `x` and `y` to build a list that represents the following expression:

```
(begin (+ 1 1) (+ 2 3))
```

What would be the result of calling `eval` on a quoted version of the expression above?

```
(eval '(begin (+ 1 1) (+ 2 3)))
```

5

Now that we know what this expression should evaluate to, let's build our scheme list.

How would we construct the scheme list for the expression `(begin (+ 1 1) (+ 2 3))` using quasiquotation?

`` `(begin ,x ,y) ``

How would we construct this scheme list using the `list` special form?

`(list 'begin x y)`

How would we construct this scheme list using the `cons` special form?

```
(cons 'begin (cons x (cons y nil)))
```

Q2: Multiple Assignment

Recall that in Scheme, the expression returned by a macro procedure is evaluated in the environment that called the macro. This concept allows us to set variables in the calling environment using calls to macro procedures! This is not possible with regular scheme procedures because any `define` expressions would be evaluated in the procedure's environment (and thus bind a symbol in that environment rather than the calling environment). In this problem, we'll explore this idea in more detail.

In Python, we can bind two variables in one line as follows:

```
>>> x, y = 1, 2
>>> x
1
>>> y
2
>>> x, y = y, x # swap the values of x and y
>>> x
2
>>> y
1
```

The expressions on the right of the assignment are first evaluated, then assigned to the variables on the left. Let's try to implement a similar feature in Scheme using macros.

Write a macro `multi-assign` which takes in two symbols `sym1` and `sym2` as well as two expressions `expr1` and `expr2`. It should bind `sym1` to the value of `expr1` and `sym2` to the value of `expr2` in the environment from which the macro was called.

```
scm> (multi-assign x y 1 (- 3 1))
scm> x
1
scm> y
2
```

First, implement a version of `multi-assign` which evaluates `expr1` first, binds it to `sym1`, then evaluates `expr2` and binds it to `sym2` (the order here is important).

```
(define-macro (multi-assign sym1 sym2 expr1 expr2)
    `(begin (define ,sym1 ,expr1) (define ,sym2 ,expr2) undefined)
)
```

Note that we use undefined as the last expression in the `begin` form so that nothing is output to the terminal.

This solution is great, but it doesn't quite behave in quite the same way that it does in Python:

```
scm> (multi-assign x y 1 (+ 2 3))
scm> x
1
scm> y
5
scm> (multi-assign x y y x)
scm> x
5
scm> y
5
```

Notice that x and y were not swapped like we wanted. This is because of the order of evaluation and bindings: first, the value of y is bound to x. Afterwards, x is evaluated and bound to y, but at this point, x no longer has its old value, it is actually the value of y!

Now, try writing a version of `multi-assign` which matches the behavior in Python, i.e. `expr1` and `expr2` should be both evaluated before being assigned to `sym1` and `sym2`.

```
scm> (multi-assign x y 5 6)
scm> x
5
scm> y
6
scm> (multi-assign x y y x)
scm> x
6
scm> y
5
```

```
(define-macro (multi-assign sym1 sym2 expr1 expr2)
    `(begin (define ,sym2 (list ,expr1 ,expr2))
            (define ,sym1 (car ,sym2))
            (define ,sym2 (car (cdr ,sym2)))
            undefined))
)
```

The idea behind this solution is to make `sym2` hold the values of *both* `expr1` and `expr2`, so that even after `sym1` gets bound to the value of `expr1`, `sym2` will have access to the previous value of `expr2` (even if `expr2` happens to be `sym1` itself, as it is in the variable swapping case). We can achieve this by binding `sym2` to a list containing the values of `expr1` and `expr2`!

To understand the motivation behind this somewhat strange looking solution, let's take a look at a different, and perhaps more intuitive, way to write this macro:

```
(define-macro (multi-assign sym1 sym2 expr1 expr2)
    `(begin (define val1 ,expr1) (define val2 ,expr2)
            (define ,sym1 val1) (define ,sym2 val2) undefined))
```

This achieves the desired behavior for the most part. It improves upon the solution in the first part of the problem by evaluating `expr1` and `expr2` before assigning them to `sym1` and `sym2`, allowing us to do variable swapping. However, there's one issue with this solution which is that it uses `val1` and `val2` to store the values of `expr1` and `expr2` in the current environment. If these symbols have already been defined in this environment (and were perhaps being used for something else), they will be overwritten!

Therefore, we need to write `multi-assign` so that it only binds symbols that were passed into the macro.

Q3: Replace

Write the macro `replace`, which takes in a Scheme expression `expr`, a Scheme symbol or number `old`, and a Scheme expression `new`. The macro replaces all instances of `old` with `new` before running the modified code.

```
(define (replace-helper e o n)
  (if (pair? e)
      (cons (replace-helper (car e) o n) (replace-helper (cdr e) o n))
      (if (eq? e o) n e)))
(define-macro (replace expr old new)
    (replace-helper expr old new))
```

Tail Recursion

When writing a recursive procedure, it's possible to write it in a **tail recursive** way, where all of the recursive calls are tail calls. A **tail call** occurs when a function calls another function as the last action of the current frame.

Consider this implementation of `factorial` that is *not* tail recursive:

```
(define (factorial n)
  (if (= n 0)
      1
      (* n (factorial (- n 1)))))
```

The recursive call occurs in the last line, but it is not the last expression evaluated. After calling `(factorial (- n 1))`, the function still needs to multiply that result with `n`. The final expression that is evaluated is a call to the multiplication function, not `factorial` itself. Therefore, the recursive call is `not` a tail call.

Here's a visualization of the recursive process for computing `(factorial 6)` :

```
(factorial 6)
(* 6 (factorial 5))
(* 6 (* 5 (factorial 4)))
(* 6 (* 5 (* 4 (factorial 3))))
(* 6 (* 5 (* 4 (* 3 (factorial 2)))))
(* 6 (* 5 (* 4 (* 3 (* 2 (factorial 1))))))
(* 6 (* 5 (* 4 (* 3 (* 2 1)))))
(* 6 (* 5 (* 4 (* 3 2))))
(* 6 (* 5 (* 4 6)))
(* 6 (* 5 24))
(* 6 120)
720
```

The interpreter first must reach the base case and only then can it begin to calculate the products in each of the earlier frames.

We can rewrite this function using a helper function that remembers the temporary product that we have calculated so far in each recursive step.

```
(define (factorial n)
  (define (fact-tail n result)
    (if (= n 0)
        result
        (fact-tail (- n 1) (* n result))))
  (fact-tail n 1))
```

`fact-tail` makes a single recursive call to `fact-tail`, and that recursive call is the last expression to be evaluated, so it is a tail call. Therefore, `fact-tail` is a tail recursive process.

Here's a visualization of the tail recursive process for computing `(factorial 6)`:

```
(factorial 6)
(fact-tail 6 1)
(fact-tail 5 6)
(fact-tail 4 30)
(fact-tail 3 120)
(fact-tail 2 360)
(fact-tail 1 720)
(fact-tail 0 720)
720
```

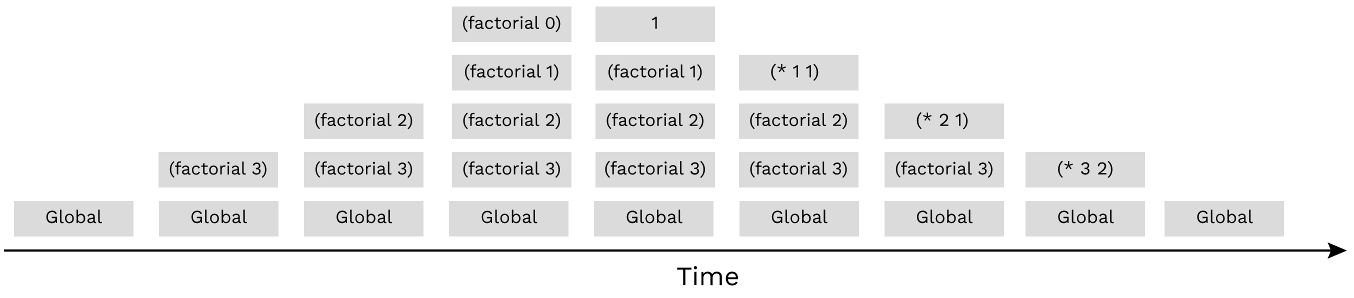
The interpreter needed less steps to come up with the result, and it didn't need to re-visit the earlier frames to come up with the final product.

In this example, we've utilized a common strategy in implementing tail-recursive procedures which is to pass the result that we're building (e.g. a list, count, sum, product, etc.) as a argument to our procedure that gets changed across recursive calls. By doing this, we do not have to do any computation to build up the result after the recursive call in the current frame, instead any computation is done *before* the recursive call and the result is passed to the next frame to be modified further. Often, we do not have a parameter in our procedure that can store this result, but in these cases we can define a helper procedure with an extra parameter(s) and recurse on the helper. This is what we did in the `factorial` procedure above, with `fact-tail` having the extra parameter `result`.

Tail Call Optimization

When a recursive procedure is not written in a tail recursive way, the interpreter must have enough memory to store all of the previous recursive calls.

For example, a call to the `(factorial 3)` in the non tail-recursive version must keep the frames for all the numbers from 3 down to the base case, until it's finally able to calculate the intermediate products and forget those frames:

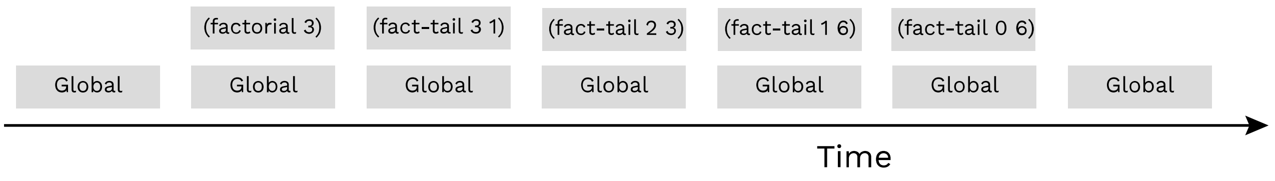


Example Tree

For non tail-recursive procedures, the number of active frames grows proportionally to the number of recursive calls. That may be fine for small inputs, but imagine calling `factorial` on a large number like 10000. The interpreter would need enough memory for all 1000 calls!

Fortunately, proper Scheme interpreters implement **tail-call optimization** as a requirement of the language specification. TCO ensures that tail recursive procedures can execute with a constant number of active frames, so programmers can call them on large inputs without fear of exceeding the available memory.

When the tail recursive `factorial` is run in an interpreter with tail-call optimization, the interpreter knows that it does not need to keep the previous frames around, so it never needs to store the whole stack of frames in memory:



Example Tree

Tail-call optimization can be implemented in a few ways:

1. Instead of creating a new frame, the interpreter can just update the values of the relevant variables in the current frame (like `n` and `result` for the `fact-tail` procedure). It reuses the same frame for the entire calculation, constantly changing the bindings to match the next set of parameters.
2. How our 61A Scheme interpreter works: The interpreter builds a new frame as usual, but then *replaces* the current frame with the new one. The old frame is still around, but the interpreter no longer has any way to get to it. When that happens, the Python interpreter does something clever: it *recycles* the old frame so that

the next time a new frame is needed, the system simply allocates it out of recycled space. The technical term is that the old frame becomes “garbage”, which the system “garbage collects” behind the programmer’s back.

Tail Context

When trying to identify whether a given function call within the body of a function is a tail call, we look for whether the call expression is in **tail context**.

Given that each of the following expressions is the last expression in the body of the function, the following expressions are tail contexts:

1. the second or third operand in an `if` expression
2. any of the non-predicate sub-expressions in a `cond` expression (i.e. the second expression of each clause)
3. the last operand in an `and` or an `or` expression
4. the last operand in a `begin` expression’s body
5. the last operand in a `let` expression’s body

For example, in the expression `(begin (+ 2 3) (- 2 3) (* 2 3))`, `(* 2 3)` is a tail call because it is the last operand expression to be evaluated.

Q4: Is Tail Call

For each of the following procedures, identify whether it contains a recursive call in a tail context. Also indicate if it uses a constant number of active frames.

```
(define (question-a x)
  (if (= x 0) 0
      (+ x (question-a (- x 1)))))
```

In the recursive case, the last expression that is evaluated is a call to `+`. Therefore, the recursive call is not in tail context, and each of the frames remain active. This procedure uses a number of active frames proportional to the input `x`.

```
(define (question-b x y)
  (if (= x 0) y
      (question-b (- x 1) (+ y x))))
```

The recursive call is the third operand in the if expression, so it is in tail context. This means that the last expression that will be evaluated in the body of this procedure is the recursive procedure call, so this procedure can be run with a constant number of active frames.

```
(define (question-c x y)
  (if (> x y)
      (question-c (- y 1) x)
      (question-c (+ x 10) y)))
```

The recursive calls are the second and third operands of the `if` expression. Only one of these calls is actually evaluated, and whichever one it is will be the last expression evaluated in the body of the procedure. This procedure therefore can be run with a constant number of active frames.

Note that if you actually try and evaluate this procedure, it will never terminate. But at least it won't crash from hitting max recursion depth!

```
(define (question-d n)
  (if (question-d n)
      (question-d (- n 1))
      (question-d (+ n 10))))
```

The second and third recursive calls are in tail context, but the first is not. Since not all the recursive calls are tail calls, this procedure requires active frames for all of the recursive calls.

Additionally, this question will actually lead to infinite recursion because the if condition will never reach a base case!

```
(define (question-e n)
  (cond ((<= n 1) 1)
        ((question-e (- n 1)) (question-e (- n 2)))
        (else (begin (print 2) (question-e (- n 3))))))
```

The second and third recursive calls are the second expressions in a clause, so they are in tail context. However, the first recursive call is not in tail context. Since not all recursive calls are tail calls, this procedure is not tail recursive and does not use a constant number of active frames.

Q5: Sum

Write a tail recursive function that takes in a Scheme list and returns the numerical sum of all values in the list. You can assume that the list contains only numbers (no nested lists).

```
scm> (sum '(1 2 3))
6
scm> (sum '(10 -3 4))
11
```

```
(define (sum lst)
  (define (sum-sofar lst current-sum)
    (if (null? lst)
        current-sum
        (sum-sofar (cdr lst) (+ (car lst) current-sum))))
  (sum-sofar lst 0)

)

; ALTERNATE SOLUTION
(define (sum lst)
    (cond
      ((null? lst) 0)
      ((null? (cdr lst)) (car lst))
      (else (sum (cons (+ (car lst) (car (cdr lst))) (cdr (cdr lst)))))
    )
)

(expect (sum '(1 2 3)) 6)
(expect (sum '(10 -3 4)) 11)
```

Video walkthrough

Q6: Reverse

Write a tail-recursive function `reverse` that takes in a Scheme list a returns a reversed copy. *Hint*: use a helper function!

```
scm> (reverse '(1 2 3))
(3 2 1)
scm> (reverse '(0 9 1 2))
(2 1 9 0)
```

```
(define (reverse lst)
  (define (reverse-tail sofar rest)
    (if (null? rest)
          sofar
          (reverse-tail (cons (car rest) sofar) (cdr rest))))
  (reverse-tail nil lst)
)

(expect (reverse '(1 2 3)) (3 2 1))
(expect (reverse '(0 9 1 2)) (2 1 9 0))
```