CS 61A
Summer 2019

Macros & Tail Calls

Discussion 9: August 1, 2019

1 Tail-Call Optimization

Scheme implements tail-call optimization, which allows programmers to write recursive functions that use a constant amount of space. A **tail call** occurs when a function calls another function as its **last action of the current frame**. In this case, the frame is no longer needed, and we can remove it from memory. In other words, if this is the last thing you are going to do in a function call, we can reuse the current frame instead of making a new frame.

Consider this implementation of `factorial`.

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

The recursive call occurs in the last line, but it is not the last expression evaluated. After calling `(fact (- 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 `fact` itself. Therefore, the recursive call is **not** a tail call.

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 (fact 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. We say that a recursive function is **tail recursive** if all of its recursive calls are tail calls.

Using a constant number of frames

Tail recursive processes can use a constant amount of memory because each recursive call frame does not need to be saved.

Our original implementation of `fact` required the program to keep each frame open because the last expression multiplies the recursive result with `n`. Therefore, at each frame, we need to remember the current value of `n`.

In contrast, the tail recursive `fact-tail` does not require the interpreter to remember the values for `n` or `result` in each frame. Instead, we can just *update* the value of `n` and `result` of the current frame! Therefore, we can keep reusing a single frame to complete this calculation.

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:

- the second or third operand in an `if` expression
- any of the non-predicate sub-expressions in a `cond` expression (i.e. the second expression of each clause)
- the last operand in an `and` or an `or` expression
- the last operand in a `begin` expression's body
- 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.

Questions

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

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


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


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


(define (question-e n)
  (cond ((= n 0) 1)
        ((question-e (- n 1)) (question-e (- n 2)))
        (else (begin (print 2) (question-e (- n 3))))))
```

1.2 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).

```
(define (sum lst)
```

1.3 Write a tail recursive function that returns the nth fibonacci number. We define fib(0) = 0 and fib(1) = 1.

```
(define (fib n)
  (define (fib-sofar ____________________________________________)

    (if __________________________________________________________

        __________________________________________________________

        (fib-sofar _______________________________________________)

  (fib-sofar ____________________________________________________))
```

2 Extra Question

2.1 Write a tail recursive function that takes in a number and a sorted list. The function returns a sorted copy with the number inserted in the correct position.

(a) Begin by writing a tail recursive function that reverses a list.

```
(define (reverse lst)
  (define (reverse-sofar lst lst-sofar)

    (if (null? lst) ______________________________________________________________________

      ___________________________________________________________________________________))

  ___________________________________________________________________________________________)
```

(b) Next, write a tail recursive function that concatenates two lists together. You may use `reverse`.

```
(define (append a b)
  (define (rev-append-tail a b)

    (if (null? a) ________________________________________________________________________

      ___________________________________________________________________________________))

  ___________________________________________________________________________________________)
```

(c) Finally, implement `insert`. You may use `reverse` and `append`.

```
(define (insert n lst)
  (define (rev-insert lst rev-lst)

    (cond ((null? lst) ______________________________________________________________)

          ((> (car lst) n) ___________________________________________________________)

          (else __________________________________________________________________________)))

  ___________________________________________________________________________________________)
```

3 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

3.1 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
```

3.2 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
```

3.3 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
```