CS 61A
Summer 2020

Tail Recursion and Macros

Discussion 11: July 30, 2020

1 Tail Recursion

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))))
```

yes

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

yes

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

yes

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

no

Note: This worksheet is a problem bank—most TAs will not cover all the problems in discussion section.

1.2 Write a tail recursive function, `reverse`, that takes in a Scheme list and returns a reversed copy.

```
(define (reverse lst)
```

```
scm> (define (reverse lst)
scm> (define (reverse-helper curr rest)
scm> (if (null? rest) curr
scm> (reverse-helper (cons (car rest) curr) (cdr rest))))
scm> (reverse-helper '() lst))

reverse

scm> (reverse '(1 2 3 4 5))

(5 4 3 2 1)
```

1.3 Write a tail recursive function, `insert`, that takes in a number and a sorted list. The function returns a sorted copy with the number inserted in the correct position. For example, `(insert 6 '(2 4 5 7))` should result in `(2 4 5 6 7)`.

```
(define (insert n lst)
```

```
scm> (define (insert n lst)
scm> (define (insert-helper curr rest)
scm> (if (null? rest) (reverse (cons n curr))
scm> (if (< n (car rest))
scm> (append curr (list n) rest)
scm> (insert-helper (append curr (list (car rest))) (cdr rest)))))
scm> (insert-helper '() lst))
scm>
scm> (insert 6 '(2 4 5 7))

insert,(2 4 5 6 7)
```

2 Macros in Python?

Imagine if we wanted to write a function in python that would create and evaluate the following line of code:

```
[<expr> for i in range(5)]
```

Where we could pass in any arbitrary expression `<expr>`, that would then be evaluated 5 times and have the results listed. This is the kind of problem that macros were made for!

A macro is a procedure that operates on unevaluated code. In the example above, `<expr>` is not necessarily an object, but could be any piece of code, such as `print("Hello")`, `[j ** 2 for j in range(i)]`, or `i >= 5`. For these expressions it is important to our problem statement that they not be evaluated until *after* they have been inserted into our list comprehension, either because they have side effects that will be apparent in the global frame, or because they depend on `i` in some way (there can be other reasons too!). So, we need a procedure that, instead of manipulating objects, manipulates code itself. This is where macros come in.

Unfortunately for us, Python does not have the ability to make true macros. Let's see what happens if we try to solve this problem with a traditional function.

```
def list_5(expr):
        return [expr for i  in range(5)]

>>> lst = list_5(print(10))
10
>>> lst
[None, None, None, None, None]
```

This isn't quite what we want. Because of Python evaluation rules, instead of evaluating a list of 5 `print(10)` statements, our `expr` was evaluated before the function was ever called, meaning 10 was only printed once.

The issue here is order of evaluation—we don't want our expression parameter to be evaluated until *after* it is inserted into our list comprehension. Even though we don't have the ability to make real macros when writing Python, we can write a function in the spirit of macros by using the `eval` function.

```
def list_5(expr):
        return eval("[" + expr + " for i  in range(5)]")

>>> lst = list_5("print(10)")
10
10
10
10
10
>>> lst
[None, None, None, None, None]
```

Now we see the number 10 printed 5 times as a side effect of our function, just like we want! We circumvented Python's evaluate-operands-before-evaluating-function body rule by passing in our desired expression as a string. Then, after we constructed our list comprehension in string form we used the `eval` function to force evaluation of the output to be the last step in the execution of this function. We were able to write code that took code as a parameter and restructured it in the form of new code!

Note: Although this is a cool hack we can do with Python, its usage is unfortunately rather limited. The `eval` function will throw a `SyntaxError` if it is passed any "compound" blocks such as `if: ...`, `while: ...`, `for: ...`, etc. (this does not include list comprehensions, or one-line if-statements such as `1 if a > b else 0`.)

Questions

2.1 Write a "macro" in Python called `make_lambda` that takes in two parameters: `params`, a string containing a comma-separated list of variable names; and `body`, a Python expression in string form, and returns a lambda function that takes `params` as its parameters and `body` as its body.

```
def make_lambda(params, body):
    """
    >>> f = make_lambda("x, y", "x + y")
    >>> f
    <function <lambda> at ...>
    >>> f(1, 2)
    3
    >>> g = make_lambda("a, b, c", "c if a > b else -c")
    >>> g(1, 2, 3)
    -3
    >>> make_lambda("f, x, y", "f(x, y)")(f, 1, 2)
    3
    """
```

```
def make_lambda(params, body):
    return eval("lambda " + params + ":" + body)
```

3

2.2 (Fun problem that is certainly *not* in scope for 61A) Python has a construction that's just like the quasiquote in Scheme! For example, the following two lines of code are equivalent:

```
>>> x = 2
>>> "The value of x is: " + x
'The value of x is 2'
>>> f"The value of x is {x}"
'The value of x is 2'
```

A string with an `f` (which is called an "f-String") is like a quasiquoted expression in Scheme, and anything in that string in braces will be treated as though it has been unquoted. Try to rewrite `make_lambda` using this syntax!

```
def make_lambda(params, body):
```

```
def make_lambda(params, body):
    return eval(f"lambda {params}: {body}")
```

3 Macros in Scheme

Now let's talk about real macros, in Scheme. 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.

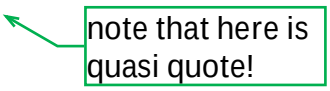
Questions

3.1 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
```

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



3.2 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)
```

```
scm>  (define-macro (prune-expr expr)
scm>    (define (prune-helper expr)
scm>      (if (null? (car expr))
scm>          '()
scm>          (if (null? (cdr expr))
scm>              expr
scm>              (cons (car expr) (prune-helper (cdr (cdr expr)))))))
scm>
scm>    `(,(car expr) ,(prune-helper (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
```

3.3 Using macros, let's make 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
```

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

```
(define-macro (when condition exprs)
  (list 'if____ _____________))
```

```
scm> (define-macro (when condition exprs)
scm> (list 'if conditon (list 'begin exprs) 'okay))
```

(b) Now, implement the macro using quasiquotes.

```
(define-macro (when condition exprs)
  `(if _ ____________))
```

```
scm>  (define-macro (when condition exprs)
scm>  `(if ,condition (begin ,exprs) `okay))
```