TAIL RECURSION, SCOPE, AND PROJECT 4 11

COMPUTER SCIENCE 61A

Noveber 12, 2012

1 Tail Recursion

Today we will look at Tail Recursion and Tail Call Optimizations in Scheme, and how they relate to iteration in Python. Let's start with an example:

From Lecture Notes: "Implementations of Scheme are required to be properly tail-recursive. This allows the execution of an iterative computation in constant space, even if the iterative computation is described by a syntactically recursive procedure."

```
def exp(b, n):
    #Theta(N) time and space
    if n == 0:
        return 1
     return b * exp(b, n-1)

def exp(b, n):
    #Theta(N) time but Theta(1) space
    total = 1
    for _ in range(n):
        total = total * b
    return total
```

But we don't have for and while iterative constructs in scheme (well we do, but we don't use them), so how do we accomplish same idea?

1.1 Optimized Tail Call

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

Note the similarities and differences of the above procedure to a python version using a while loop.

```
def factorial(n, k):
    while n > 0:
        n, k = n-1, k*n
    return k
```

So how exactly is this an optimization, let's look at how the two different versions of scheme factorial (one recursive, and one tail recursive), and how they grow in space.

Procedure Growth in Space (from Wikipedia)
Recursive:

```
 call factorial (3)
   3 * call factorial(2)
    3 * 2 * call factorial(1)
     3 * 2 * 1 * call factorial(0)
      3 * 2 * 1 * 1
    3 * 2 * 1
   3 * 2
 return 6
```

More efficient tail recursive, in terms of both space and time:

```
 call factorial (3)
   call fact (3 1)
   call fact (2 3)
   call fact (1 6)
   call fact (0 6)
  return 6
```

Note that the optimization we are witnessing here is not an optimization in time, but in space. In most examples in this class we have looked at time growth of a procedure, but a procedure can also be analyzed by how much space it uses either in memory, cache, or CPU. These topics will be explored more in depth CS61C. Here, both these functions (one recursive, one tail recursive), both are linear recursions that both take linear time.

But the optimization comes in that the tail recursive version uses constant space (or space that does not grow with respect to the input), where as the recursive version also grows linearly in space with respect to its input.

Exact Definition of Tail Recursive Calls (from lecture) :
A procedure call that has not yet returned is active. Some procedure calls are tail calls. A Scheme interpreter should support an unbounded number of active tail calls. A tail call is a call expression in a tail context:

**Note: A call expression is not a tail call if more computation is still required in the calling procedure. Linear recursions can often be re-written to use tail calls.

1.2 Questions

1. Write a function **`last`**, which takes in a list and returns the last element of the list:

   ```
   (define (last s))
   ```

2. Write the function that returns the nth Fibonacci number (try writing recursive first, then think tail recursive).

   ;Tree-recursive version:

   ```
   (define (fib-tree n)
   ```

   ;Tail-recursive version:

   ```
   (define (fib n)
   ```

3. Write a function that inserts number n into a sorted list of numbers, s. (once again try writing recursive version first, then think tail-recursive).

;Non-tail-recursive version

```
(define (insert-non-tail n s)
```

; From lecture: reverse-iter

```
(define (reverse-iter s result)
  (if (null? s) result
    (reverse-iter (cdr s) (cons (car s) result))))
```

;Tail-recursive version (Hint: Use a helper function)

```
(define (insert n s)
```

Note: Not all recursive procedures can be rewritten in a tail recursive pattern, and the tricky part is understanding which ones can't, for instance the subset-sum problem.

2 Lexical Scope Vs Dynamic Scope

To this point in the class, we have only talked about Lexical Scoping. Remember that the scoping rule is the environment diagram rule that decides what a new frame's parent is when the frame is created. So when a procedure is called, a new frame is created, that frame needs a parent, and whatever scoping rule you are using will decide what that

parent frame is. Basically, this rule determines what will be in scope for that new frame, or what variables it will be able to see.

While some languages (not many) let you choose which scoping to use when defining a variable, most use lexical. However, dynamic is usually a good step towards fully understanding lexical scoping, and so it is good to learn both. (You will have to implement both in your project).

- Lexical scope: The parent of a frame is the environment in which a procedure was defined.
- Dynamic scope: The parent of a frame is the environment in which a procedure was called

Lexical Scoping also relates heavily to closures, a closure (in simplest terms) is a procedure that is also bound with the environment frame it was defined in, this is necessary so that the procedure will know what frame to extend when it is called, otherwise, it would not be able to tell. This is why in your project, and lambda has a frame instance variable.

While python is lexically scoped, you will need to be able to draw environment diagrams using both scoping rules, and it is possible that a question like this could show up on the final!

2.1 Problems

1. Draw the environment diagram for the following code, first with Lexical Scoping, and then with Dynamic Scoping. :

```
x = 10
def foo(x):
    return x * bar(2)
def bar(y):
    return y + x
foo(3)
```

2.
```
def make_counter(x):
    def count(m):
        nonlocal x
        if m == 'inc':
            x += 1
        elif m== 'count':
            return x
    return count
c1 = make_counter(5)
c2 = make_counter(7)
c1('inc')
c1('inc')
c1('count')
```

3 Project 4: Scheme Interpreter

For the rest of discussion, we will discuss your project at a high level, this is a chance to really look at the code, and go over as a group what is going on with the skeleton code you have been given for the project, and the ideas you are expected to implement, please ask questions.

For your project you need to fill in the missing pieces of the scheme interpreter. This involves completing the missing aspects of the read-eval-print loop. You will implement a few basic functions to parse scheme expressions for the reader, and then besides that most of the project will be spent working in the eval phase.

Eval, for most complex interpreters, involves a loop of its own, called the eval-apply loop. To this point, our interpreters have been simple. In our calculator interpreter, we called eval, which called apply, which finished. But now that we want to have user defined procedures, which have their own bodies which need to be evaluated, so apply may now also need to call eval to interpret an answer. This will be an example of Mutual Recursion.

There will also be more bells and whistles wrapped around this which you will have to figure out, but for now lets look at the meat of the code for the project. Some comments have been included to help you understand what is going on. Your TA will explain this code.

```
def scheme_eval(expr, env):
    """Evaluate Scheme expression EXPR in environment ENV."""

   #The expression must not be none
    if expr is None:
        raise SchemeError("Cannot evaluate an undefined expression.")

    # Evaluate Atoms
   """If the atom is a symbol (either a primitive procedure symbol,
   or variable, look it up!)"""
    if scheme_symbolp(expr):
        return env.lookup(expr)
   # if its primitive (number, etc), simply return
    elif scheme_atomp(expr):
        return expr

    # All non-atomic expressions are lists.
    if not scheme_listp(expr):
        raise SchemeError("malformed list: {0}".format(str(expr)))
    first, rest = expr.first, expr.second
```

```
    #Special Form Evaluation Comes Next
    # Evaluate Combinations
    if first in LOGIC_FORMS:
        return scheme_eval(LOGIC_FORMS[first](rest, env), env)
    #evaluate lambda special form
    elif first == "lambda":
        return do_lambda_form(rest, env)
   #evaluate mu special form
    elif first == "mu":
        return do_mu_form(rest)
   #evaluate define special form
    elif first == "define":
        return do_define_form(rest, env)
   #evaluate quote special form
    elif first == "quote":
        return do_quote_form(rest)
   #evaluate let special form
    elif first == "let":
        expr, env = do_let_form(rest, env)
        return scheme_eval(expr, env)
   #not a special form, apply normal scheme rules of evaluation
    else:
       """evaluate the operator (in case its a variable, this is
       different from calculator)"""
        procedure = scheme_eval(first, env)
       #map scheme_eval to args, eval in current env
        args = rest.map(lambda operand: scheme_eval(operand, env))
       #apply procedure to evaluated args
        return scheme_apply(procedure, args, env)

def scheme_apply(procedure, args, env):
    """Apply Scheme PROCEDURE to argument values
    ARGS in environment ENV."""
   #handle primitive, happen by magic through underlying python
    if isinstance(procedure, PrimitiveProcedure):
        return apply_primitive(procedure, args, env)
   #handle lambdas, use lexical scope
    elif isinstance(procedure, LambdaProcedure):
        "*** YOUR CODE HERE ***"
    #handle mus, use dynamic scope
```

```
    elif isinstance(procedure, MuProcedure):
        "*** YOUR CODE HERE ***"
    else:
        raise SchemeError("Cannot call {0}".format(str(procedure)))

#Primitives dont create a new env,  get applied in current env
def apply_primitive(procedure, args, env):
    """Apply PrimitiveProcedure PROCEDURE to a
    Scheme list of ARGS in ENV."""
```

Try and answer the following questions, there is not an exact answer to these questions that we are looking for, but rather this is to get you to think about interpreters, and `scheme-eval`, you can just describe your answers. These are questions you will have to answer in your project in order to write the code.

3.1 Questions

1. Why does `scheme-eval` check in order of atoms, symbols, special forms, then normal application? What would go wrong if it didn't?

2. Why does the operator to `scheme-eval` also have to be evaluated (unlike calculator), what could the operator be?

3. When a procedure is called, which frame are its arguments evaluated in?

4. Why does `scheme-eval` pass the current env to `scheme-apply`?

5. Should `scheme-apply`'s implementation call `scheme-eval`? If yes then where?

6. What frame do primitive procedures get evaluated in? Should they create a new frame? Should lambdas or mus?

7. True or False: All define does is point variables to values in the current frame, it does not actually make a procedure.

3.2 Frames

Now lets look at Frames, and the differences between Primitives, Lambdas, and Mus Each class keeps tracks of the items that are necessary for it to behave properly. Each Frame

maintains a dictionary of bindings of variables to values, and a parent frame, which can be None (for instance for the global frame). The Frame also supports methods to lookup variables starting from its given location on its environment chain, and to add variables to its bindings.

The Lambda and Mu classes are for User Defined Procedures, they keep track of formals, which is a list of argument names, and a body, which will be another scheme expression. The difference between the two is that lambdas also keep track of the environment they were created in, why? We will answer that in a minute.

There is also PrimitiveProcedures which are neither lambdas nor mus, they are builtin and do not require a new frame to be evaluated. This is because for our purposes, as always, we dont care how primitives happen, we just assume they happen magically. In general, no frames are created for the application of primitives, but some may need a frame, and so we have the option to pass a frame in, and you will deal with these cases in the project. For our interpreter we will let underlying python handle the scheme primitives. So then a PrimitiveProcedure in our interpreter is an instance of a class that maintains the underlying python function it uses, an env if necessary, plus a static list of all Primitives.

Here is the skeleton code from the project for the Frame, LambaProcedure, MuProcedure, and PrimitiveProcedure classes.

```
class Frame(object):
    """An environment frame binds Scheme symbols to Scheme values."""

    def __init__(self, parent):
        """An empty frame with a PARENT frame (that may be None)."""
        self.bindings = {}
        self.parent = parent

    def lookup(self, symbol):
        """Return the value bound to SYMBOL.
                        Errors if SYMBOL is not found."""
        raise SchemeError("unknown identifier: {0}".format(str(symbol)))

    def global_frame(self):
        """The global environment at the root of the parent chain."""
        e = self
        while e.parent is not None:
            e = e.parent
        return e

    def make_call_frame(self, formals, vals):
```

```
        """Return a new local frame whose parent is SELF,
        in which the symbols in the Scheme formal parameter
        list FORMALS are bound to the Scheme
        values in the Scheme value list VALS. """
        frame = Frame(self)
        return frame

    def define(self, sym, val):
        """Define Scheme symbol SYM to have value VAL in SELF."""
        self.bindings[sym] = val

class LambdaProcedure(object):
    """A procedure defined by a lambda expression
        or the complex define form."""

    def __init__(self, formals, body, env):
        """A procedure whose formal parameter list is
        FORMALS (a Scheme list), whose body is the single
        Scheme expression BODY, and whose parent
        environment is the Frame ENV.  A lambda expression
        containing multiple expressions, such as
        (lambda (x) (display x) (+ x 1)) can be handled by using
        (begin (display x) (+ x 1)) as the body."""
        self.formals = formals
        self.body = body
        self.env = env

class MuProcedure(object):
    """A procedure defined by a mu expression,
       which has dynamic scope."""

    def __init__(self, formals, body):
        """A procedure whose formal parameter list is
        FORMALS (a Scheme list), whose body is the
        single Scheme expression BODY.  A mu expression
        containing multiple expressions, such as
        (mu (x) (display x) (+ x 1)) can be handled by using
        (begin (display x) (+ x 1)) as the body."""
        self.formals = formals
        self.body = body
```

```
class PrimitiveProcedure:
    """A Scheme procedure defined as a Python function."""

    def __init__(self, fn, use_env=False):
        self.fn = fn
        self.use_env = use_env

_PRIMITIVES = []
```

Once again, here are some high level questions to answer about this code, and to get you thinking about the project if you havent already.

3.3 Questions

1. Write pseudocode or a description of how you would implement lookup for frames.

2. When would `make_call_frame` be used? How does this relate to Mus?

3. Why do lambdas have an env but mus don't, when would you use the env instance variable in lambdas?