TAIL CALLS AND INTERPRETERS 8

COMPUTER SCIENCE 61A

November 5, 2015

1 Tail Calls

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. As the tail call is the last action for the current frame, Scheme won't make any further variable lookups in the frame. Therefore, the frame is no longer needed, and we can remove it from memory.

Consider this version of `factorial` that does *not* use tail calls:

```
(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.

However, 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 prod)
    (if (= n 0) prod
        (fact-tail (- n 1) (* n prod))))
  (fact-tail n 1))
```

`fact-tail` makes single recursive call that is the last expression to be evaluated, so it is a tail call. Therefore, `fact-tail` is a tail recursive process. Tail recursive processes can take 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 `prod` in each frame. Once a new recursive frame is created, the old one can be dropped, as all the information needed is included in the new frame. Therefore, we can carry out the calculation using only enough memory for a single frame.

1.1 Identifying tail calls

A function call is a tail call if it is in a **tail context**. We consider the following to be tail contexts:

- the last sub-expression in a lambda's body
- the second or third sub-expression in an `if` form
- any of the non-predicate sub-expressions in a `cond` form
- the last sub-expression in an `and` or an `or` form
- the last sub-expression in a `begin`'s body

2 Questions

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

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

(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))))
```

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

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

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

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

4. 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.

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

3 Calculator

We are beginning to dive into the realm of interpreting computer programs – that is, writing programs that understand other programs. In order to do so, we'll have to examine programming languages in-depth. The *Calculator* language, a subset of Scheme, will be the first of these examples.

In today's discussion, we'll be implementing Calculator using regular Python.

The Calculator language is a Scheme-syntax language that includes only the four basic arithmetic operations: $+$, $-$, $*$, and $/$. These operations can be nested and can take varying numbers of arguments. Here's a few examples of Calculator in action:

```
> (+ 2 2)
4

> (- 5)
-5

> (* (+ 1 2) (+ 2 3))
15
```

Our goal now is to write an interpreter for this Calculator language. The job of an interpreter is to evaluate expressions. So, let's talk about expressions. A Calculator expression is just like a Scheme list. To represent Scheme lists in Python, we use `Pair` objects. For example, the list `(+ 1 2)` is represented as `Pair('+', Pair(1, Pair(2, nil)))`. The `Pair` class is similar to the Scheme procedure `cons`, which would represent the same list as `(cons '+ (cons 1 (cons 2 nil)))`.

`Pair` is very similar to `Link`, the class we developed for representing linked lists. In addition to `Pair` objects, we include a `nil` object to represent the empty list. Both `Pair` instances and `nil` have methods:

1. `__len__`, which returns the length of the list.
2. `__getitem__`, which allows indexing into the pair.
3. `map`, which applies a function, `fn`, to all of the elements in the list.
4. `to_py_list`, which returns a Python list with the same elements.

Here's an implementation of what we described:

```
class nil:
    """The empty list"""
    def __len__(self):
        return 0
    def map(self, fn):
        return self

nil = nil() # this hides the nil class *forever*

class Pair:
    def __init__(self, first, second=nil):
        self.first, self.second = first, second

    def __len__(self):
        n, second = 1, self.second
        while isinstance(second, Pair):
            n, second = n + 1, second.second
        if second is not nil:
            raise TypeError("length attempted on improper list")
        return n

    def __getitem__(self, k):
        if k == 0:
            return self.first
        if k < 0:
            raise IndexError("negative index into list")
        elif self.second is nil:
            raise IndexError("list index out of bounds")
        elif not isinstance(self.second, Pair):
            raise TypeError("ill-formed list")
        return self.second[k-1]

    def map(self, fn):
        """Returns a Scheme list after applying Python function
        fn over self."""
        applied = fn(self.first)
        if self.second is nil or isinstance(self.second, Pair):
            return Pair(applied, self.second.apply_to_all(fn))
        else:
            raise TypeError("ill-formed list")
```

```
    def to_py_list(self):
        """Returns a Python list containing the elements of this
        Scheme list."""
        y, result = self, []
        while y is not nil:
            result.append(y.first)
            if not isinstance(y.second, Pair) and y.second is not
                nil:
                raise TypeError("ill-formed list")
            y = y.second
        return result
```

4 Questions

1. Translate the following Calculator expressions into calls to the `Pair` constructor.
   ```
   > (+ 1 2 (- 3 4))
   ```

   ```
   > (+ 1 (* 2  3) 4)
   ```

2. Draw the box and pointer diagram and translate the following Python representation of Calculator expressions into the proper Scheme syntax:
   ```
   >>> Pair('+', Pair(1, Pair(2, Pair(3, Pair(4, nil)))))
   ```

   ```
   >>> Pair('+', Pair(1, Pair(Pair('*', Pair(2, Pair(3, nil))),
      nil)))
   ```

5 Evaluation

Evaluation discovers the form of an expression and executes a corresponding evaluation rule.

We'll go over two such expressions now:

1. *Primitive* expressions are evaluated directly. e.g. "1" just evaluates to itself.
2. *Call* expressions are evaluated in the same way you've been doing them by hand all semester:
   (1) **Evaluate** the operator.
   (2) **Evaluate** the operands from left to right.
   (3) **Apply** the operator to the operands.

Here's `calc_eval`:

```
def calc_eval(exp):
    if not isinstance(exp, Pair): # primitive expression
        return exp
    else: # call expression

        # Step 1: evaluate the operator.
        operator = exp.first

        # Step 2: evaluate the operands.
        operands = exp.second
        args =  operands.map(calc_eval).to_py_list()

    # Step 3: apply the operator to the operands.
    return calc_apply(operator, args)
```

How do we apply the operator? We'll dispatch on the operator name with `calc_apply`:

```
def calc_apply(operator, args):
    if operator == '+':
        return sum(args)
    elif operator == '-':
        if len(args) == 1:
            return -args[0]
        else:
            return args[0] - sum(args[1:])
    elif operator == '*':
        return reduce(mul, args, 1)
```

Depending on what the operator is, we can match it to a corresponding Python call. Each conditional clause above handles the application of one operator.

Something very important to keep in mind: `calc_eval` deals with **expressions** (in Calculator), `calc_apply` deals with **values** (which are in Python).

6 Questions

1. Suppose we typed each of the following expressions into the Calculator interpreter. How many calls to `calc_eval` would they each generate? How many calls to `calc_apply`?

   ```
   > (+ 2 4 6 8)
   ```

   ```
   > (+ 2 (* 4 (- 6 8)))
   ```

2. Alyssa P. Hacker and Ben Bitdiddle are also tasked with implementing the `and` operator, as in `(and (= 1 2) (< 3 4))`. Ben says this is easy: they just have to follow the same process as in implementing `*` and `/`. Alyssa is not so sure. Who's right?

3. Now that you've had a chance to think about it, you decide to try implementing `and` yourself. You may assume the conditional operators (e.g. `<`, `>`, `=`, etc) have already been implemented for you.