TAIL CALLS, ITERATORS, AND GENERATORS 11

COMPUTER SCIENCE 61A

November 13, 2014

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-iter n prod)
    (if (= n 0) prod
        (fact-iter (- n 1) (* n prod))))
  (fact-iter n 1))
```

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

1.2 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))))
```

1.3 Extra Questions

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

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

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

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

2 Iterators

An *iterator* is an object that represents a sequence of values. Here is an example of a class that implements Python's iterator interface. This iterator calculates all of the natural numbers one-by-one, starting from zero:

```
class Naturals():
    def __init__(self):
        self.current = 0
    def __next__(self):
        result = self.current
        self.current += 1
        return result
    def __iter__(self):
        return self
```

There are two components of Python's iterator interface: the `__next__` method, and the `__iter__` method.

2.1 `__next__`

The `__next__` method usually does two things:

1. calculates the next value
2. checks if it has any values left to compute

To return the next value in the sequence, the iterator does some computation defined in the `__next__` method.

When there are no more values left to compute, the `__next__` method must raise a type of exception called `StopIteration`. This signals the end of the sequence.

Note: the `__next__` method defined above does NOT raise any `StopIteration` exceptions. Why? Because there are always more values left to compute! Remember, there is no "last natural number", so there is technically no "end of the sequence." However, if you wanted to define a *finite* iterator, then you would raise a `StopIteration` after returning the final value.

2.2 `__iter__`

The purpose of the `__iter__` method is to return an iterator object. By definition, an iterator object is an object that has implemented both the `__next__` and `__iter__` methods.

This has an interesting consequence. If a class implements both a `__next__` method and a `__iter__` method, its `__iter__` method can just return `self` (like in the example). Since the class implements both `__next__` and `__iter__`, it is technically an iterator object, so its `__iter__` method can just return itself.

2.3 Implementation

When defining an iterator object, you should always keep track of how much of the sequence has already been computed. In the above example, we use an instance variable `self.current` to keep track.

Iterator objects maintain state. Successive calls to `__next__` will most likely output different values each time, so `__next__` is considered *non-pure*.

How do we call `__next__` and `__iter__`? Python has built-in functions called `next` and `iter` for this. Calling `next(some_iterator)` will then cause Python to implicitly call `some_iterator`'s `__next__` method. Calling `iter(some_iterator)` will make a similar implicit call to `some_iterator`'s `__iter__` method.

For example, this is how we would use the `Naturals` iterator:

```
>>> nats = Naturals()
>>> nats_iter = iter(nats)
>>> next(nats_iter)
0
>>> next(nats_iter)
1
>>> next(nats_iter)
2
```

However, we don't really need to call `iter` on `nats`. Why not?

Because you can use iterator objects in `for` loops. In other words, any object that satisfies the iterator interface can be iterated over:

```
>>> nats = Naturals()
>>> for n in nats:
        print(n)
0
1
2
...   # Forever!
```

This works because the Python `for` loop implicitly calls the `__iter__` method of the object being iterated over, and repeatedly calls `next` on it. In other words, the above interaction is (basically) equivalent to:

```
nats_iter = iter(nats)
is_done = False
while not is_done:
    try:
        val = next(nats_iter)
        print(val)
    except StopIteration:
        is_done = True
```

2.4 Questions

1. Define an iterator whose i-th element is the result of combining the i-th elements of two input iterables using some binary operator, also given as input. The resulting iterator should have a size equal to the size of the shorter of its two input iterators.

```
>>> from operator import add
>>> evens = IterCombiner(Naturals(), Naturals(), add)
>>> next(evens)
0
>>> next(evens)
2
>>> next(evens)
4

class IterCombiner(object):
    def __init__(self, iter1, iter2, combiner):




    def __next__(self):




    def __iter__(self):
```

2. What is the result of executing this sequence of commands?

```
>>> naturals = Naturals()
>>> doubled_naturals = IterCombiner(naturals, naturals, add)
>>> next(doubled_naturals)
```

```
>>> next(doubled_naturals)
```

2.5 Extra Practice

1. Create an iterator that generates the sequence of Fibonacci numbers.

```
class Fibonacci(object):
    def __init__(self):




    def __next__(self):




    def __iter__(self):
```

3 Generators

A **generator** function is a special kind of Python function that uses a `yield` statement instead of a `return` statement to report values. When a generator function is called, it returns an iterable object.

Here is an iterator for the natural numbers written using the generator construct:

```
def generate_naturals():
    current = 0
    while True:
        yield current
        current += 1
```

Calling `generate_naturals()` will return a generator object:

```
>>> gen = generate_naturals()
>>> gen
<generator object gen at ...>
```

To use the generator object, you then call `next` on it:

```
>>> next(gen)
0
>>> next(gen)
1
>>> next(gen)
2
```

Think of a generator object as containing an implicit `__next__` method. This means, by definition, a generator object is an iterator.

3.1 yield

The `yield` statement is similar to a `return` statement. However, while a `return` statement closes the current frame after the function exits, a yield statement causes the frame to be saved until the next time `__next__` is called, which allows the generator to automatically keep track of the iteration state.

Once `__next__` is called again, execution picks up from where the previously executed `yield` statement left off, and continues until the next `yield` statement (or the end of the function) is encountered.

Including a `yield` statement in a function automatically signals to Python that this function will create a generator. When we call the function, it will return a generator object, instead of executing the code inside the body. When the returned generator's `__next__` method is called, the code in the body is executed for the first time, and stops executing upon reaching the first `yield` statement.

3.2 Implementation

Because generators are technically iterators, you can implement `__iter__` methods using only generators. For example,

```
class Naturals():
    def __init__(self):
        self.current = 0
    def __iter__(self):
        while True:
            yield self.current
            self.current += 1
```

Naturals `__iter__` method now returns a generator object. The usage of a `Naturals` object is exactly the same as before:

```
>>> nats = Naturals()
>>> nats_iter = iter(nats)
>>> next(nats_iter)
0
>>> next(nats_iter)
1
>>> next(nats_iter)
2
```

There are a couple of things to note:

- *No `__next__` method in `Naturals`*. Remember, `__iter__` only needs to return an object that has implemented a `__next__` method. Since generators have their own `__next__` method, the new `Naturals` implementation is perfectly valid.
- *`nats` is a `Naturals` object and `nats_iter` is a generator*

Since generators are iterators, you can also use generators in `for` loops.

3.3 Questions

1. Define a generator that yields the sequence of perfect squares.

    ```
    def perfect_squares():
    ```

3.4 Extra Practice

1. Write a generator function that returns lists of all subsets of the positive integers from 1 to n. Each call to this generator's `__next__` method will return a list of subsets of the set `[1, 2, ..., n]`, where `n` is the number of times `__next__` was previously called.

```
def generate_subsets():
    """
    >>> subsets = generate_subsets()
    >>> next(subsets)
    [[]]
    >>> next(subsets)
    [[], [1]]
    >>> next(subsets)
    [[], [1], [2], [1, 2]]
    """
```