1 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, was the first of these examples.

The Calculator language is a Scheme-syntax language that currently includes only the four basic arithmetic operations: +, −, ∗, and /. These operations can be nested and can take varying numbers of arguments. A few examples of calculator in action are given on the right. 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 the same as 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, except that the second attribute doesn’t have to be a linked list. In addition to Pair objects, we include a nil object to represent the empty list. Pair instances 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.

nil has the methods __len__, __getitem__, and map.

Here’s an implementation of what we described:

```python
class nil:
    """Represents the special empty pair nil in Scheme."""
    def __repr__(self):
        return 'nil'
    def __len__(self):
        return 0
    def __getitem__(self, i):
        raise IndexError('Index out of range')
    def map(self, fn):
        return nil

nil = nil()  # this hides the nil class *forever*```
class Pair:
    """Represents the built-in pair data structure in Scheme."""
    def __init__(self, first, second):
        self.first = first
        self.second = second
    def __repr__(self):
        return 'Pair({}, {}).format(self.first, self.second)
    def __len__(self):
        return 1 + len(self.second)
    def __getitem__(self, i):
        if i == 0:
            return self.first
        return self.second[i-1]
    def map(self, fn):
        return Pair(fn(self.first), self.second.map(fn))

Questions

1.1 Translate the following Calculator expressions into calls to the Pair constructor.
    Hint: in the example from earlier, the list (+ 1 2) is represented as Pair('+', Pair(1, Pair(2, nil)) ).

> (+ 1 2 (- 3 4))

>>> Pair('+', Pair(1, Pair(2, Pair(  
                   Pair('-', Pair(3, Pair(4, nil))), nil)))))

> (+ 1 (* 2 3) 4)

>>> Pair('+', Pair(1, Pair(Pair('*', Pair(2, Pair(3, nil))),  
                        Pair(4,nil))))

Video walkthrough

1.2 Translate the following Python representations of Calculator expressions into the proper Scheme syntax:

>>> Pair('+', Pair(1, Pair(2, Pair(3, Pair(4, nil))))))

> (+ 1 2 3 4)

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

> (+ 1 (* 2 3))

Video walkthrough
2 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. For example, the numbers 3.14 and 165 just evaluate to themselves, and the string “+” evaluates to the `calc_add` function.

2. **Call expressions** are evaluated in the same way you’ve been doing them 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`:

```python
def calc_eval(exp):
    """Evaluates a Calculator expression represented as a Pair."""
    if isinstance(exp, Pair):
        return calc_apply(calc_eval(exp.first),
                           list(exp.second.map(calc_eval)))
    elif exp in OPERATORS:
        return OPERATORS[exp]
    else: # Atomic expressions
        return exp
```

And here’s `calc_apply`:

```python
def calc_apply(op, args):
    """Applies an operator to a Pair of arguments."""
    return op(*args)
```

The `*args` syntax expands a list of arguments. For example:

```python
>>> calc_apply(print, [1, 2, 3]) # Becomes print(1, 2, 3), not print([1, 2, 3])
1 2 3
```

Video walkthrough

Questions

2.1 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)
6 calls to eval: 1 for the entire expression, and then 1 for each operator and operand.  
1 call to apply the addition operator.  

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

10 calls to eval: 1 for the whole expression, then 1 for each of the operators and operands. When we encounter another call expression, we have to evaluate the operators and operands inside as well. 

3 calls to apply each of the operators.  

Video walkthrough

2.2 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?  

Alyssa. We can’t handle `and` in the apply step since `and` is a special form: it is short-circuited. We need to create a special case for it in `calc_eval`.  

2.3 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.  

```python
def calc_eval(exp):
    if isinstance(exp, Pair):
        if exp.first == 'and':
            return eval_and(exp.second)
        else:
            return calc_apply(calc_eval(exp.first), list(exp.second.map(calc_eval)))
    elif exp in OPERATORS:
        return OPERATORS[exp]
    else: # Atomic expression
        return exp

def eval_and(operands):
    curr = operands
    last = True
    while curr is not nil:
        last = calc_eval(curr.first)
        if last is False:
            return False
        curr = curr.second
    return last
```
3 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 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.

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` 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 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.
3.1 Identifying tail calls

A function call is a tail call if it is in a tail context. However, it might not be a recursive tail call, as we saw earlier in fact which did multiplication instead. Tail context simply means the expression is the last to be evaluated in that form.

For example, 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

These make sense intuitively; for if, consider that the last expression to be evaluated in an if form is not the condition, but rather either the second or third sub-expressions which are evaluated depending on if the condition is True or False. You should be able to provide a similar reasoning for the other tail contexts listed above.

Before we jump into questions, a quick tip for defining tail recursive functions is to use helper functions. A helper function should have all the arguments from the parent function, plus additional arguments like total or counter or result.

Video walkthrough
Questions

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

In the recursive case, the last expression that is evaluated is a call to +. Therefore, the recursive call is not in a tail context, and each of the frames remain active. This function uses $\Theta(n)$ frames.

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

The if form is in a tail context, and the recursive call is the third subexpression, so it is also in a tail context. Therefore, the last evaluated expression is the recursive function call. This function therefore uses $\Theta(1)$ frames.

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

The recursive calls are the second and third sub-expressions of the if form. Therefore, only one of the calls is actually evaluated, and it is the last expression evaluated in each recursive call. This function therefore uses $\Theta(1)$ frames.

Note that if you actually try and evaluate this function, it will never terminate. But at least it won’t crash from hitting max recursion depth!

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

Both of the recursive calls are in a tail context, so they are both tail calls. However, the if predicate, (question-d n), is another recursive call. This recursive call is not in a tail context, so this function does not use a constant number of frames.

Video walkthrough

3.2 Write a tail recursive function that returns the $n$th fibonacci number. We define $\text{fib}(0) = 0$ and $\text{fib}(1) = 1$.

```
(define (fib n)
  
  (define (fib-helper n a b)
    (if (= n 0)
        a
        (fib-helper (- n 1) b (+ a b))))

  (fib-helper n 0 1))
```
Video walkthrough

3.3 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 is well-formed and contains only numbers (no nested lists).

(define (sum lst)
  (define (sum-sofar lst current-sum)
    (if (null? lst)
        current-sum
        (sum-sofar (cdr lst) (+ (car lst) current-sum))))
  (sum-sofar lst 0))

Video walkthrough

3.4 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)
        lst-sofar
        (reverse-sofar (cdr lst) (cons (car lst) lst-sofar)))
  (reverse-sofar lst '()))

Video walkthrough
(define (reverse lst)
  (define (reverse-sofar lst lst-sofar)
    (if (null? lst)
      lst-sofar
      (reverse-sofar (cdr lst) (cons (car lst) lst-sofar)))
    (reverse-sofar lst nil))

(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)
      b
      (rev-append-tail (cdr a) (cons (car a) b)))
    (rev-append-tail (reverse a) b))

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

(define (insert n lst)
  (define (rev-insert lst rev-lst)
    (cond ((null? lst) (cons n rev-lst))
      ((> (car lst) n) (rev-insert (cdr lst) (cons (car lst) rev-lst)))
      (else (rev-insert (cdr lst) (cons (car lst) rev-lst))))
    (reverse (rev-insert lst nil)))