CS 61A Fall 2018 — Interpreters & Tail Calls

Discussion 9: October 31, 2018

1 Calculator

An interpreter is a program that understands other programs. Today, we will explore how to interpret a simple language that uses Scheme syntax called *Calculator*.

```
calc> (+ 2 2)
4

calc> (- 5)
-5

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

The Calculator language includes only the four basic arithmetic operations: $+$, $-$, $*$, and $/$. These operations can be nested and can take any numbers of arguments. A few examples of calculator expressions and their corresponding values are given on the right. Recall that the reader component of an interpreter parses input strings and represents them as data structures in the implementing language. In this case, we need to represent Calculator expressions as Python objects. To represent numbers, we can just use Python numbers. To represent the names of the arithmetic procedures, we can use Python strings (e.g. `'+'`).

Call expressions are a bit more complicated. First, note that like Scheme call expressions, call expressions in Calculator look just like Scheme lists. For example, to construct the expression `(+ 2 3)` in Scheme, we would do the following:

```
scm> (cons '+ (cons 2 (cons 3 nil)))
(+ 2 3)
```

To represent Scheme lists in Python, we will use the `Pair` class. A `Pair` instance holds exactly two elements. Accordingly, the `Pair` constructor takes in two arguments, and to make a list we must nest calls to the constructor and pass in `nil` as the second element of the last pair. Note that in our implementation, `nil` is bound to a special user-defined object that represents an empty list, whereas `nil` in Scheme is actually an empty list.

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

Each `Pair` instance has two instance attributes: `first` and `second`, which are bound to the first and second elements of the pair respectively.

```
>>> p = Pair('+', Pair(2, Pair(3, nil)))
>>> p.first
'+'
>>> p.second
Pair(2, Pair(3, nil))
>>> p.second.first
2
```

Here's an implementation of what we described:

```
class Pair:
    """Represents the built-in pair data structure in Scheme."""
    def __init__(self, first, second):
        self.first = first
        self.second = second

    def map(self, fn):
        """Maps fn to every element in a well-formed list, returning a new
        Pair.

        >>> Pair(1, Pair(2, Pair(3, nil))).map(lambda x: x * x)
        Pair(1, Pair(4, Pair(9, nil)))
        """
        assert isinstance(self.second, Pair) or self.second is nil, \
            "Second element in pair must be another pair or nil"
        return Pair(fn(self.first), self.second.map(fn))

    def __getitem__(self, i):
        """Allows us to index into well-formed lists and treat well-formed
        lists like Python iterables.

        >>> p = Pair(1, Pair(2, Pair(3, nil)))
        >>> p[1]
        2
        >>> list(p)
        [1, 2, 3]
        """
        assert isinstance(self.second, Pair) or self.second is nil, \
            "Second element in pair must be another pair or nil"
        if i == 0:
            return self.first
        return self.second[i - 1]

    def __repr__(self):
        return 'Pair({}, {})'.format(self.first, self.second)

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

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

Questions

1.1 Write out the Calculator expression with proper syntax that corresponds to the following `Pair` constructor calls. Also, draw out a box and pointer diagram corresponding to each input.

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

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

1.2 Answer the following questions about a `Pair` instance representing the Calculator expression `(+ (- 2 4) 6 8)`.

i. Write out the Python expression that returns a `Pair` representing the given expression, and draw a box and pointer diagram corresponding to it.

ii. What is the operator of the call expression? Given that the `Pair` you constructed in the previous part was bound to the name `p`, how would you retrieve the operator?

iii. What are the operands of the call expression? Given that the `Pair` you constructed in Part (i) was bound to the name `p`, how would you retrieve a list containing all of the operands? How would you retrieve only the first operand?

2 Evaluation

The evaluation component of an interpreter determines the type of an expression and executes corresponding evaluation rules.

Here are the evaluation rules for the three types of Calculator expressions:

1. **Numbers** are self-evaluating. For example, the numbers 3.14 and 165 just evaluate to themselves.
2. **Names** are looked up in the `OPERATORS` dictionary. Each name (e.g. `'+'`) is bound to a corresponding function in Python that does the appropriate operation on a list of numbers (e.g. `sum`).
3. **Call expressions** are evaluated the same way you've been doing them all semester:

   (1) **Evaluate** the operator, which evaluates to a function.

   (2) **Evaluate** the operands from left to right.

   (3) **Apply** the function to the value of the operands.

The function `calc_eval` takes in a Calculator expression represented in Python and implements each of these rules:

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

Note that `calc_eval` is recursive! In order to evaluate call expressions, we must call `calc_eval` on the operator and each of the operands.

The *apply* step in the Calculator language is straight-forward, since we only have primitive procedures. This step is more complex when it comes to applying Scheme procedures, which may include user-defined procedures.

Given the Python function that implements the appropriate Calculator operation and a Python list of numbers, the `calc_apply` function simply calls the function on the arguments, and regular Python evalutation rules take place.

```
def calc_apply(fn, args):
    """Applies a Calculator operation to a list of numbers."""
    return fn(args)
```

Questions

2.1 How many calls to `calc_eval` and `calc_apply` would it take to evaluate each of the following Calculator expressions?

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

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

2.2 Suppose we want to add handling for comparison operators `>`, `<`, and `=` as well as `and` expressions to our Calculator interpreter. These should work the same way they do in Scheme.

```
calc> (and (= 1 1) 3)
3
calc> (and (+ 1 0) (< 1 0) (/ 1 0))
#f
```

i. Are we able to handle expressions containing the comparison operators (such as `<`, `>`, or `=`) with the existing implementation of `calc_eval`? Why or why not?

ii. Are we able to handle `and` expressions with the existing implementation of `calc_eval`? Why or why not?

iii. Now, complete the implementation below to handle `and` expressions. You may assume the conditional operators (e.g. `<`, `>`, `=`, etc) have already been implemented for you.

```
def calc_eval(exp):
    if isinstance(exp, Pair):
        if ________________________: # and expressions
            return eval_and(exp.second)
        else:                        # Call expressions
            return calc_apply(calc_eval(exp.first), list(exp.second.map(calc_eval)))
    elif exp in OPERATORS:           # Names
        return OPERATORS[exp]
    else:                            # Numbers
        return exp

def eval_and(operands):
```

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 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, we consider the tail context of each expression to be:

- 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

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)))))


(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))))


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

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

```
(define (fib n)
  (define (fib-sofar ___________________________________________)

    (if _________________________________________________________

        _________________________________________________________

        (fib-sofar ______________________________________________)

  (fib-sofar ____________________________________________________))
```

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)
```

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) _______________________________________________________________________

      _____________________________________________________________________________________))

  _________________________________________________________________________________________)
```

(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) _________________________________________________________________________

      _____________________________________________________________________________________))

  _________________________________________________________________________________________)
```

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

```
(define (insert n lst)
  (define (rev-insert lst rev-lst)

    (cond ((null? lst) ____________________________________________________________________)

          ((> (car lst) n) ________________________________________________________________)

          (else ___________________________________________________________________________)))

  _________________________________________________________________________________________)
```