Lecture 19: Tail Calls

CS 61A - Summer 2024

Raymond Tan

Scheme: Review

Scheme Fundamentals

- Scheme programs consist of **expressions**
  - Every line of code must evaluate to something!
- **Primitive Expressions**
  - numbers, booleans, functions, and names
  - e.g. `3.3`, `2`, `#t` <true>, `#f` <false>, `+`, `quotient`
- **Combinations**
  - groups of expressions wrapped in a set of parentheses
  - e.g. `(quotient 10 2)`, `(not #t)`
  - Combinations are always either a **call expression** or a **special form**

Call Expressions

- Always follow exactly the same syntax
- An operator first, followed by one or more operands, all wrapped in parentheses
- This is slightly different from Python, where the operator is *outside* of the parentheses

```
scm> (quotient 10 2)
5
scm> (quotient (+ 8 7) 5)
3
scm> (+ (* 3
           (+ (* 2 4)
              (+ 3 5)))
         (+ (- 10 7)
            6))
```

Special Forms

All combinations that are not call expressions are **special forms**

Call expression evaluation is always exactly the same:

(1) evaluate operator

(2) evaluate operands

(3) apply

Special forms are distinct because they, in various ways, are not evaluated by following this process

`if`

Scheme has a special form called `if`, that is very similar to Python's `if` statement

```
(if <predicate> <consequent> <alternative>)
```

The `if` special form follows this evaluation process:

- Evaluate `predicate`
- If `predicate` is truthy, evaluate and return `consequent`
- Otherwise, evaluate and return `alternative`

```
scm> (if (>= 4 5) 1 -1)
-1
scm> (if (even? 10) (+ 1 2) 1)
3
```

Scheme Lists

- All lists in Scheme are linked lists!!!

```
scm> (cons 1 (cons 2 (cons 3 nil)))
(1 2 3)
scm> (define x (cons 1 (cons 2 (cons 3 nil))))
x
scm> (car x) ; get the first element of x
1
scm> (cdr x) ; get the rest element of x
(2 3)
scm> (length x)
3
scm> (list 1 2 3) ; another way to construct a list
(1 2 3)
scm> '(1 2 3) ; another way to construct a list
(1 2 3)
```

Problem: Add to All

add-to-all takes in a parameter lst, which is a list of numbers, and returns a new list where each number is increased by 1

```
(define (add-to-all lst)
    (if (_____)
           __________
            __________))
(add-to-all '(1 2 3 4))
; expect (2 3 4 5)
```

Things to remember:

- Scheme doesn't have iteration, we need to use recursion
- We won't ever be doing mutation in Scheme, so focus on constructing a new list

Problem: Add to All (Solution)

```
(define (add-to-all lst)
   (if (null? lst)
        nil
         (cons (+ 1 (car lst))
                (add-to-all (cdr lst)))))
```

Recursion in Scheme

- Important for today's topic: Only recursion is available for us to use in Scheme, no iteration using for/while loops
    - As we know, recursion can be inefficient since a new frame must be opened for every recursive call in our environment diagrams
- *How can we make Scheme recursive implementations efficient?*

Tail Recursion

Recursion and Iteration in Python

- Consider the following implementations of `factorial(n, k)`, where:

```
factorial(n, k) = n! * k
```

```python
def factorial(n, k):
    if n == 0:
        return k
    else:
        return factorial(n-1, k*n)
```

Recursive

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

Iterative

Recursion and Iteration in Python

What is the time and space efficiency of each of these implementations?

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

Time: **Linear/O(N)**
Space: **Constant/O(1)**

```
def factorial(n, k):
    if n == 0:
        return k
    else:
        return factorial(n-1, k*n)
```

Time: **Linear/O(N)**
Space: **Linear/O(N)**

factorial(n, k) in Scheme

- Let's try to implement this function in Scheme!

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

How can we make this ***recursive*** implementation in Scheme use the same amount of resources as the ***iterative*** implementation in Python?

Demo: factorial(n, k) in Scheme

Tail Calls

- 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 using only a **constant** amount of space.
- A tail call is a call expression in a **tail context**:
  - The last body sub-expression in a `lambda` expression (or procedure definition)
  - Sub-expressions 2 & 3 in a tail context `if` expression
  - All non-predicate sub-expressions in a tail context `cond`
  - The last sub-expression in a tail context `and`, `or`, `begin`, or `let`

Tail Calls

- A tail call is a call expression in a tail context:
  - The last body sub-expression in a `lambda` expression (or procedure definition)
  - Sub-expressions 2 & 3 in a tail context `if` expression
  - All non-predicate sub-expressions in a tail context `cond`
  - The last sub-expression in a tail context `and`, `or`, `begin`, or `let`

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

The recursive call to factorial is sub-expression 3 of the if statement, and the if statement is in a tail context, therefore this recursive call is a tail call!

Example: length-of-list

- Problem: Implement a function, `length-of-list`, which takes in a list `s` and returns the length of the list.
  - The built-in function `length` does this for us already, but we want to implement our own!

length-of-list: Solution #1

```
(define (length-of-list s)
  (if (null? s) 0
    (+ 1 (length-of-list (cdr s)))))
```

Is this implementation tail recursive?

length-of-list: Solution #2

```
(define (length-of-list-tail s)
  (define (length-iter s n)
    (if (null? s) n
        (length-iter (cdr s)
                     (+ 1 n))))
  (length-iter s 0))
```

This implementation uses ***constant*** space.

Break

Tail-recursive or not?

```
;; Return whether s contains v.
(define (contains s v)
  (if (null? s)
      false
      (if (= v (car s))
          true
          (contains (cdr s) v))))
```

Yes ✅

Tail-recursive or not?

```
;; Return whether s has any repeated elements
(define (has-repeat s)
  (if (null? s)
      false
      (if (contains? (cdr s) (car s))
          true
          (has-repeat (cdr s)))))
```

Yes ✅

Tail-recursive or not?

```
;; Return the nth Fibonacci number.
(define (fib n)
  (define (fib-iter current k)
    (if (= k n)
        current
        (fib-iter (+ current
                     (fib (- k 1)))
                  (+ k 1))))
  (if (= 1 n) 0 (fib-iter 1 2)))
```

No ❌

Map and Reduce

Reduce

- Problem: Implement a function `reduce`, which takes in three parameters: `procedure`, `s`, and `start`. We apply the `procedure` onto all elements in `s,` beginning with `start`.

```
(reduce * (list 3 4 5) 2) ; -> 120
```

reduce: Implementation

```scheme
;; Reduce s using procedure and start value.
(define (reduce procedure s start)
  (if (null? s) start
    (reduce procedure
            (cdr s)
            (procedure start (car s)))))
```

Map

- Problem: Implement a function `map`, which takes in two parameters: `procedure` and `s`. We return a new version of the list `s`, where `procedure` has been applied onto each element of `procedure`.

```
(map (lambda (x) (+ x 1)) (list 3 4 5)) ; -> (4 5 6)
```

map: Solution #1

```scheme
;; Map procedure over s.
(define (map-rec procedure s)
  (if (null? s) nil
    (cons (procedure (car s))
          (map-rec procedure (cdr s)))))
```

map: Solution #2

```scheme
;; Map procedure over s.
(define (map procedure s)
  (define (map-reverse s m)
    (if (null? s) m
      (map-reverse (cdr s)
                   (cons (procedure (car s)) m))))
  (reverse (map-reverse s nil)))
```

```scheme
;; Return a copy of s with elements in reverse order.
(define (reverse s)
  (define (reverse-iter s r)
    (if (null? s) r
      (reverse-iter (cdr s)
                    (cons (car s) r))))
  (reverse-iter s nil))
```

Summary

- In Scheme, all iteration must be done through recursion
  - No for/while loops exist!
- Recursion is slow/inefficient since each recursive call opens up a new frame in our environment diagrams
- To make recursive implementations more efficient, we can implement functions using tail recursion
  - These are recursive calls in a ***tail context***