Lecture 20: Scheme (pt 2)

CS 61A - Summer 2024
Raymond Tan

Tail Calls: Review

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)
  - Examples (non-exhaustive list):
    - 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 Recursion

- Recursive calls that are in a tail context make a recursive function **tail recursive**
- In a tail-optimized language, this means that the problem can be solved in a **constant amount of space**
  - Previous frames do not have to be kept around
  - Scheme is a tail-optimized language, Python is not

Example: Factorial, non tail-recursive

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

Example: Factorial, tail-recursive

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

Dynamic Scope

Lexical Scope and Dynamic Scope

- The way in which names are looked up in Scheme and Python is called **lexical scope**
  - Definition: The parent of a frame is the frame in which a function is **defined**
- Dynamic Scope is another lookup method, where the parent of a frame is the frame in which a function is **called**
  - Under this lookup method, the parent of a function can vary!

Evaluation under Lexical Scope

```
(define f (lambda (x) (+ x y)))

(define g (lambda (x y) (f (+ x x))))

(g 3 7)
```

An error would be thrown here! Under lexical scoping, the parent of `f` is global, and the variable `y` in the body of `f` would not be found.

Evaluation under Dynamic Scope

```
(define f (lambda (x) (+ x y)))

(define g (lambda (x y) (f (+ x x))))

(g 3 7)
```

Under dynamic scoping, y in the body of f would evaluate to the value of y in g! (since we called f in the body of g). This would output 13.

Lexical vs Dynamic Scoping - Which to use?

- The original version of Lisp uses dynamic scoping
- Majority of languages use lexical scoping
- Lexically scoped languages are easier to understand (from a human perspective), since you can see exactly what is and isn't in scope just by looking at the code
- You will be implementing the `mu` special form in the Scheme project, which is a special function type that uses dynamic scoping

Miscellaneous Scheme Features

undefined

- In Scheme, the equivalent to the `None` datatype is `undefined`
- `undefined` is outputted when we print an expression
- Remember that the only falsey value in Scheme is `#f`, so `undefined` has a truthy value!

Quotation

- Quotations in Scheme allow us to keep a portion of Scheme code unevaluated
  - Very similar to Strings in Python
- We can quote primitive expressions, as well as combinations
  - Everything inside of a combination will also stay unevaluated
    - Ex: `'(1 2 (+ 1 2))` would output `(1 2 (+ 1 2))`
- Quoting can be done by using the `quote` special form or just ‘ as a shorthand

Quotation with lists

A last way we can construct lists in Scheme is with the `quote` special form—`quote` takes in an argument and returns that argument *without evaluating it*

```
scm> (define a 1)
a
scm> a
1
scm> (quote a)
a
scm> (list 1 2 3)
(1 2 3)
scm> (quote (1 2 3))
(1 2 3)
scm> 'a ; shorthand for quote
a
scm> '(1 2 3) ; can also work for combinations!
(1 2 3)
```

Quasiquotation

- Quasiquotation is a special form of quotation that allows us to unquote expressions within a quoted expression
- The quasiquote symbol is the backtick, usually located at the top left of your keyboard
  - To unquote, use a comma before an expression within a quasiquote

Quasiquotation: Demo

eval on Quoted Expressions

- eval is a built-in Scheme function which takes in a quoted expression, and evaluates that expression as Scheme code
- Notice that *call expressions are just scheme lists* where the operator is the first element in the list, and the operands are all elements that come after it

eval: Demo

Break

Practice: even-subsets

even-subsets

- Problem: Given an integer list s, we want to find all the non-empty subsets of s that have an **even** sum.
  - Ex: `(even-subsets '(1 3 2 5))`

```
-> ((2) (3 5) (3 2 5) (1 5) (1 2 5) (1 3 2) (1 3))
```

Solution

```
;;; non-empty subsets of integer list s that have an even sum
(define (even-subsets s)
  (if (null? s) nil
      (append (even-subsets (cdr s))
              (map (lambda (t) (cons (car s) t))
               (if (even? (car s))
               (even-subsets (cdr s))
               (odd-subsets (cdr s))))
          (if (even? (car s)) (list (list (car s))) nil))))

;;; non-empty subsets of integer list s that have an odd sum
(define (odd-subsets s)
  (if (null? s) nil
      (append (odd-subsets (cdr s))
              (map (lambda (t) (cons (car s) t))
               (if (odd? (car s))
               (even-subsets (cdr s))
               (odd-subsets (cdr s))))
          (if (odd? (car s)) (list (list (car s))) nil))))
```

Summary

- Two different types of scoping: Lexical vs Dynamic scoping
  - Lexical scoping: Parent of a function is the frame in which it is defined
  - Dynamic scoping: Parent of a function is the frame in which a function is called
    - Will be implementing this in the Scheme project
- Quasiquotation allows us to unquote expressions within a quoted expression
- Can call eval on quoted expressions that are valid Scheme code