CS 61A
Summer 2026

Interpreters, Macros, Tail Calls

Discussion 10: July 30, 2026

Interpreters

An interpreter is a program that allows you to interact with the computer in a certain language. It understands the expressions that you type in through that language, and performs the corresponding actions in some way, usually using an underlying language.

When we talk about an interpreter, there are two languages at work:

1. **The language being interpreted/implemented.**
2. **The underlying implementation language.**

Many interpreters use a Read-Eval-Print Loop (REPL). This loop waits for user input, and then processes it in three steps:

- **Read:** The interpreter takes the user input (a string) and turns the user input string into small pieces (tokens), and then takes the tokens and organizes them into data structures.
- **Eval:** Mutual recursion between eval and apply evaluate the expression to obtain a value.

Representing Lists

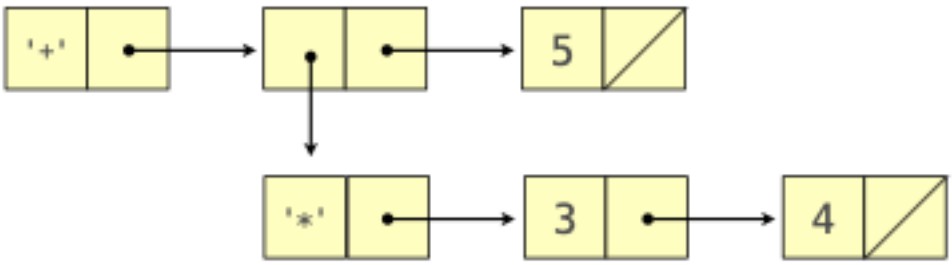
A Scheme call expression is a Scheme list that is represented using a `Link` instance in Python.

For example, the call expression `(+ (* 3 4) 5)` is represented as:

```
Link('+', Link(Link('*', Link(3, Link(4, nil))), Link(5, nil)))
```

Those `nil`'s are optional because `nil` is `Link.empty`, which is the default second argument to the `__init__` method of the `Link` class.

```
Link('+', Link(Link('*', Link(3, Link(4))), Link(5)))
```



(+ (* 3 4) 5)

The `Link` class and `nil` object are defined in link.py of the Scheme project.

Evaluation

Q1: Evaluation

Which of the following are evaluated when `scheme_eval` is called on `(if (< x 0) (- x) (if (= x -2) 100 y))` in an environment in which `x` is bound to -2? Select all that apply. (Assume `<`, `-`, and `=` have their default values.)

- `if`
- `<`
- `=`
- `x`
- `y`
- `0`
- `-2`
- `100`
- `-`
- `(`
- `)`

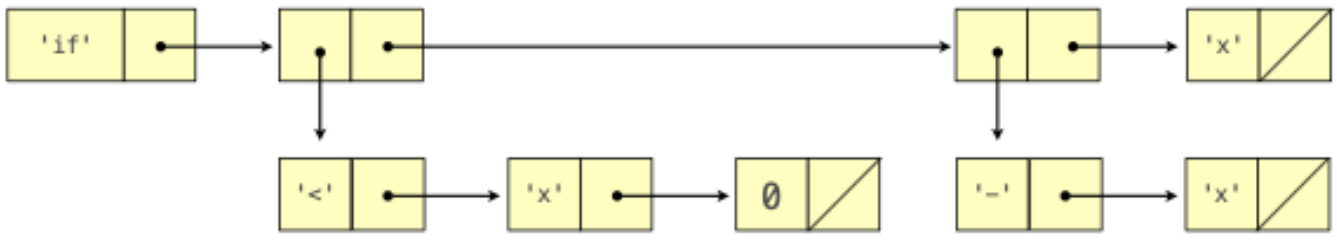
To evaluate the expression `(+ (* 3 4) 5)` using the Project 4 interpreter, `scheme_eval` is called on the following expressions (in this order):

1. `(+ (* 3 4) 5)`
2. `+`
3. `(* 3 4)`
4. `*`
5. `3`
6. `4`
7. `5`

An `if` expression is also a Scheme list represented using a `Link` instance.

For example, `(if (< x 0) (- x) x)` is represented as:

```
Link('if', Link(Link('<', Link('x', Link(0, nil))), Link(Link('-', Link('x', nil)), Link('x', nil))))
```



To evaluate this expression in an environment in which `x` is bound to 2 (and `<` and `-` have their default values), `scheme_eval` is called on the following expressions (in this order):

1. `(if (< x 0) (- x) x)`
2. `(< x 0)`
3. `<`
4. `x`
5. `0`
6. `x`

The symbol `if` is not evaluated because it is the start of a special form, not part of a call expression. The symbols that introduce special forms (`and`, `if`, `lambda`, etc.) are never evaluated.

The symbol `-` is not evaluated, nor is the whole sub-expression `(- x)` that it appears in, because `(< x 0)` evaluates to `#f`. **If you're still not certain why some parts are evaluated and some aren't, ask the course staff.**

Q2: Print Evaluated Expressions

Define `print_evals`, which takes a Scheme expression `expr` that contains only numbers, `+`, `*`, and parentheses. It prints all of the expressions that are evaluated during the evaluation of `expr`. They are printed in the order that they are passed to `scheme_eval`.

Note: Calling `print` on a `Link` instance will print the Scheme expression it represents.

```
>>> print(Link('+', Link(Link('*', Link(3, Link(4, nil))), Link(5, nil))))
(+ (* 3 4) 5)
```

```
def print_evals(expr):
    """Print the expressions that are evaluated while evaluating expr.
    expr: a Scheme expression containing only (, ), +, *, and numbers.
    >>> nested_expr = Link('+', Link(Link('*', Link(3, Link(4, nil))), Link(5, nil)))
    >>> print_evals(nested_expr)
    (+ (* 3 4) 5)
    +
    (* 3 4)
    *
    3
    4
    5
    >>> print_evals(Link('*', Link(6, Link(7, Link(nested_expr, Link(8, nil))))))
    (* 6 7 (+ (* 3 4) 5) 8)
    *
    6
    7
    (+ (* 3 4) 5)
    +
    (* 3 4)
    *
    3
    4
    5
    8
    """
    if not isinstance(expr, Link):
        print(expr)

    else:
        print(expr)
        while expr is not nil:
            print_evals(expr.first)
            expr = expr.rest
```

Macros!

A macro is a code transformation that is created using `define-macro` and applied using a call expression. A macro call is evaluated by:

1. Binding the formal parameters of the macro to the unevaluated operand expressions of the macro call.
2. Evaluating the body of the macro, which returns an expression.
3. Evaluating the expression returned by the macro in the environment of the original macro call.

```
scm> (define-macro (twice expr) (list 'begin expr expr))
twice
scm> (twice (+ 2 2))  ; evaluates (begin (+ 2 2) (+ 2 2))
4
scm> (twice (print (+ 2 2)))  ; evaluates (begin (print (+ 2 2)) (print (+ 2 2)))
4
4
```

Q3: Mystery Macro

Figure out what this `mystery-macro` does. Try to describe what it does by reading the code and discussing examples as a group.

```
(define-macro (mystery-macro expr old new)
    (mystery-helper expr old new))

(define (mystery-helper e o n)
  (if (and (list? e) (not (null? e)))
      (cons (mystery-helper (car e) o n) (mystery-helper (cdr e) o n))
      (if (eq? e o) n e)))
```

Here are some example uses of `mystery-macro` that could help you understand what it does and how it might be used.

```
scm> (define five 5)
five
scm> (mystery-macro (* x x) x five)
25
scm> (mystery-macro (* x x) x (+ five 1))
36
scm> (mystery-macro '(* x x) x y)
(* y y)
scm> (mystery-macro (> (x) (> (y) (+ x y))) > lambda)
(lambda (x) (lambda (y) (+ x y)))
scm> (mystery-macro (begin e e e) e (print five))
5
5
5
```

The `mystery-macro` replaces all instances of an `old` symbol with a `new` expression before evaluating the expression `expr`.

Q4: Arg Repeater

First, write the procedure `repeater`, which takes in expression `expr` and a non-negative integer `k`, and outputs a new list with `expr` repeated `k` times.

Then write the macro `arg-repeater`, which takes in a function `fn`, an expression `expr` and a non-negative integer `k`, which corresponds to the number of arguments that `fn` takes in. `arg-repeater` passes `k` copies of `expr` to a call to `fn`.

Note: The skeleton code is just a suggestion; feel free to use your own structure if you prefer.

```
(define (repeater expr k)
  ; Returns a list with k copies of expr
  (if (= 0 k)
      nil
      (cons expr (repeater expr (- k 1)))))


(expect (repeater 2 0) ())
(expect (repeater 2 5) (2 2 2 2 2))

(define-macro (arg-repeater fn expr k)
  ; Fills in fn with k copies of expr
  (cons fn (repeater expr k)))


(define (add-four a b c d) (+ a b c d))
(define (mult-one x) x)
(define (mult-seven a1 a2 a3 a4 a5 a6 a7)
  (* a1 a2 a3 a4 a5 a6 a7))

(expect (arg-repeater add-four 10 4) 40)
(expect (arg-repeater mult-one -2 1) -2)
; 2 multiplied to itself 7 times = 2^7 = 128
(expect (arg-repeater mult-seven 2 7) 128)
```

Tail Calls

Q5: Reverse

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

```
scm> (reverse '(1 2 3))
(3 2 1)
scm> (reverse '(0 9 1 2))
(2 1 9 0)
```

```
(define (reverse lst)
  (define (reverse-tail sofar rest)
    (if (null? rest)
          sofar
          (reverse-tail (cons (car rest) sofar) (cdr rest))))
  (reverse-tail nil lst)
)

(expect (reverse '(1 2 3)) (3 2 1))
(expect (reverse '(0 9 1 2)) (2 1 9 0))
```

More Scheme

Q6: Slice It!

Implement the `get-slicer` procedure, which takes integers `a` and `b` and returns an *a-b slicing function*. An `a-b` slicing function takes in a list as input and outputs a new list with the values of the original list from index `a` (inclusive) to index `b` (exclusive).

Your implementation should behave like Python slicing, but should assume a step size of one with no negative slicing indices. Indices start at zero.

Note: the skeleton code is just a suggestion. Feel free to use your own structure if you prefer.

```
(define (get-slicer a b)
  (define (slicer lst)
    (define (slicer-helper c i j)
      (cond
        ((or (null? c) (<= j i)) nil)
        ((= i 0) (cons (car c) (slicer-helper (cdr c) i (- j 1))))
        (else (slicer-helper (cdr c) (- i 1) (- j 1)))))
    (slicer-helper lst a b))
  slicer)

; DOCTESTS (No need to modify)
(define a '(0 1 2 3 4 5 6))
(define one-two-three (get-slicer 1 4))
(define one-end (get-slicer 1 10))
(define zero (get-slicer 0 1))
(define empty (get-slicer 4 4))

(expect (one-two-three a) (1 2 3))
(expect (one-end a) (1 2 3 4 5 6))
(expect (zero a) (0))
(expect (empty a) ())
```