CS 61A Summer 2025

Scheme Lists, Interpreters

Discussion 9: July 29, 2025

Scheme Lists

> As you read through this section, it may be difficult to understand the differences between the various representations of Scheme containers. We recommend that you use our online Scheme interpreter to see the box-and-pointer diagrams of pairs and lists that you're having a hard time visualizing! (Use the command `(autodraw)` to toggle the automatic drawing of diagrams.)

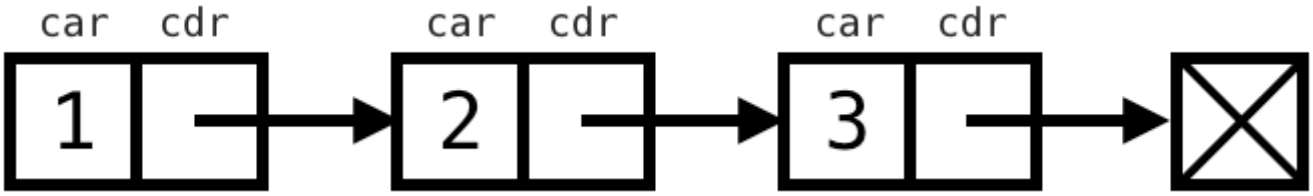
Lists

Scheme lists are very similar to the linked lists we've been working with in Python. Just like how a linked list is constructed of a series of `Link` objects, a Scheme list is constructed with a series of pairs, which are created with the constructor `cons`.

Scheme lists require that the `cdr` is either another list or `nil`, an empty list. A list is displayed in the interpreter as a sequence of values (similar to the `__str__` representation of a `Link` object). For example,

```
scm> (cons 1 (cons 2 (cons 3 nil)))
(1 2 3)
```

Here, we've ensured that the second argument of each `cons` expression is another `cons` expression or `nil`.



list

We can retrieve values from our list with the `car` and `cdr` procedures, which now work similarly to the Python `Link`'s `first` and `rest` attributes. (Curious about where these weird names come from? Check out their etymology.)

```
scm> (define a (cons 1 (cons 2 (cons 3 nil))))  ; Assign the list to the name a
a
scm> a
(1 2 3)
scm> (car a)
1
scm> (cdr a)
(2 3)
scm> (car (cdr (cdr a)))
3
```

If you do not pass in a pair or nil as the second argument to `cons`, it will error:

```
scm> (cons 1 2)
Error
```

`list` Procedure

There are a few other ways to create lists. The `list` procedure takes in an arbitrary number of arguments and constructs a list with the values of these arguments:

```
scm> (list 1 2 3)
(1 2 3)
scm> (list 1 (list 2 3) 4)
(1 (2 3) 4)
scm> (list (cons 1 (cons 2 nil)) 3 4)
((1 2) 3 4)
```

Note that all of the operands in this expression are evaluated before being put into the resulting list.

Quote Form

We can also use the quote form to create a list, which will construct the exact list that is given. Unlike with the `list` procedure, the argument to `'` is *not* evaluated.

```
scm> '(1 2 3)
(1 2 3)
scm> '(cons 1 2)           ; Argument to quote is not evaluated
(cons 1 2)
scm> '(1 (2 3 4))
(1 (2 3 4))
```

Built-In Procedures for Lists

There are a few other built-in procedures in Scheme that are used for lists. Try them out in the interpreter!

```
scm> (null? nil)               ; Checks if a value is the empty list
True
scm> (append '(1 2 3) '(4 5 6)) ; Concatenates two lists
(1 2 3 4 5 6)
scm> (length '(1 2 3 4 5))     ; Returns the number of elements in a list
5
```

Q1: Pair Up

Implement `pair-up`, which takes a list `s`. It returns a list of lists that together contain all of the elements of `s` in order. Each list in the result should have 2 elements. The last one can have up to 3.

```
;;; Return a list of pairs containing the elements of s.
;;;
;;; scm> (pair-up '(3 4 5 6 7 8))
;;; ((3 4) (5 6) (7 8))
;;; scm> (pair-up '(3 4 5 6 7 8 9))
;;; ((3 4) (5 6) (7 8 9))
(define (pair-up s)
    (if (<= (length s) 3)
        (list s)
        (cons (list (car s) (car (cdr s))) (pair-up (cdr (cdr s))))
    ))

(expect (pair-up '(3 4 5 6 7 8)) ((3 4) (5 6) (7 8)) )
(expect (pair-up '(3 4 5 6 7 8 9)) ((3 4) (5 6) (7 8 9)) )
```

Q2: List Insert

Write a Scheme function that, when given an element, a list, and an index, inserts the element into the list at that index. You can assume that the index is in bounds for the list.

```
(define (insert element lst index)
    (if (= index 0)
        (cons element lst)
        (cons (car lst) (insert element (cdr lst) (- index 1))))
)

(expect (insert 2 '(1 7 9) 2) (1 7 2 9))

(expect (insert 'a '(b c) 0) (a b c))
```

Q3: Interleave

Implement the function `interleave`, which takes two lists `lst1` and `lst2` as arguments. `interleave` should return a list that interleaves the elements of the two lists. (In other words, the resulting list should contain elements alternating between `lst1` and `lst2`, starting at `lst1`).

If one of the input lists to `interleave` is shorter than the other, then `interleave` should alternate elements from both lists until one list has no more elements, and then the remaining elements from the longer list should be added to the end of the new list. If `lst1` is empty, you may simply return `lst2` and vice versa.

```
(define (interleave lst1 lst2)
    (if (or (null? lst1) (null? lst2))
        (append lst1 lst2)
        (cons (car lst1)
                (cons (car lst2)
                    (interleave (cdr lst1) (cdr lst2))))))

; Alternate Solution
    (cond
        ((null? lst1) lst2)
        ((null? lst2) lst1)
        (else (cons (car lst1) (interleave lst2 (cdr lst1))))
    )
)
```

The base cases for both solutions (which are equivalent), follow directly from the spec. That is, if we run out of elements in one list, then we should simply append the remaining elements from the longer list.

The first solution constructs the interleaved list two elements at a time, by `cons`-ing together the first two elements of each list alongside the result of recursively calling `interleave` on the `cdr`'s of both lists.

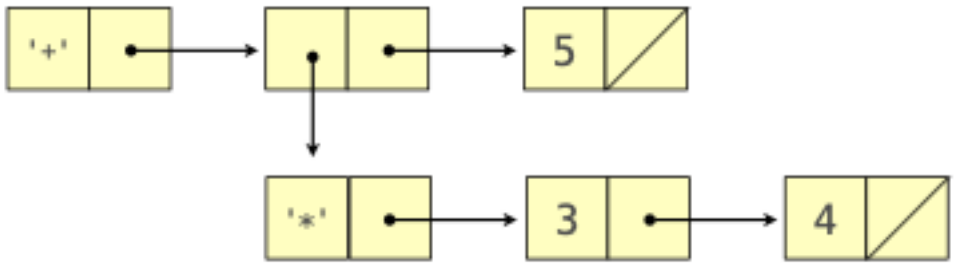
The second solution constructs the interleaved list one element at a time by swapping which list is passed in for `lst1`. Thus, we can then grab elements from only `lst1` to construct the list.

Scheme Call Expressions

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

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

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



(+ (* 3 4) 5)

The `Pair` class and `nil` object are defined in pair.py of the Scheme project.

```
class Pair:
    "A Scheme list is a Pair in which rest is a Pair or nil."
    def __init__(self, first, rest):
        self.first = first
        self.rest = rest

    ... # There are also __str__, __repr__, and map methods, omitted here.
```

Q4: Representing Expressions

Write the Scheme expression in Scheme syntax represented by each `Pair` below. Try drawing the linked list diagram too.

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

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

```
>>> Pair('and', Pair(Pair('<', Pair(1, Pair(0, nil))), Pair(Pair('/', Pair(1, Pair(0, nil
    ))), nil)))
```

Answer 1: `(+ (* 3 4) 5)`

Answer 2: `(+ 1 (* 2 3))`

Answer 3: `(and (< 1 0) (/ 1 0))`

Evaluation

To evaluate the expression `(+ (* 3 4) 5)` using the 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`

The `*` is evaluated because it is the operator sub-expression of `(* 3 4)`, which is an operand sub-expression of `(+ (* 3 4) 5)`.

By default, `*` evaluates to a procedure that multiplies its arguments together. But `*` could be redefined at any time, and so the symbol `*` must be evaluated each time it is used in order to look up its current value.

```
scm> (* 2 3)  ; Now it multiplies
6
scm> (define * +)
*
scm> (* 2 3)  ; Now it adds
5
```

Q5: 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? (Assume `<`, `-`, and `=` have their default values.)

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

The expression is `(if (< x 0) (- x) (if (= x -2) 100 y))`.

1. We look at the `if` first

However, we do not evaluate `if` since it is a special form. Special forms are looked up and not evaluated (See Scheme Project). Given x = -2, we then evaluate the condition `(< x 0)`:

2. As usual, we evaluate the operator first, then the operands
3. So we evaluate `<` and then we evaluate `x` and `0`
4. `(< -2 0)` evaluates to true

Since the condition is true, we evaluate the return value `(- x)`:

5. We evaluate the operator `-` and then we evaluate `x`
6. `x` evaluates to `-2`
7. Now we have `(- -2)` so this evaluates to 2.

Since the condition `(< x 0)` is true, the alternative `(if (= x -2) 100 y)` is not evaluated.

So, the elements that are evaluated in the process are: `<, x, 0, -`

Q6: 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 `Pair` instance will print the Scheme expression it represents.

```
>>> print(Pair('+', Pair(Pair('*', Pair(3, Pair(4, nil))), Pair(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 = Pair('+', Pair(Pair('*', Pair(3, Pair(4, nil))), Pair(5, nil)))
        >>> print_evals(nested_expr)
        (+ (* 3 4) 5)
        +
        (* 3 4)
        *
        3
        4
        5
        >>> print_evals(Pair('*', Pair(6, Pair(7, Pair(nested_expr, Pair(8, nil))))))
        (* 6 7 (+ (* 3 4) 5) 8)
        *
        6
        7
        (+ (* 3 4) 5)
        +
        (* 3 4)
        *
        3
        4
        5
        8
        """
        if not isinstance(expr, Pair):
            print(expr)
        else:
            print(expr)
            while expr is not nil:
                print_evals(expr.first)
                expr = expr.rest
```

Challenge

Q7: 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) ())
```

Submit Attendance

You're done! Excellent work this week. Please be sure to ask your section TA for the attendance form link and fill it out for credit. (one submission per person per section).