CS 61A Spring 2026

Interpreters

Discussion 10: April 15, 2026

Representing Lists

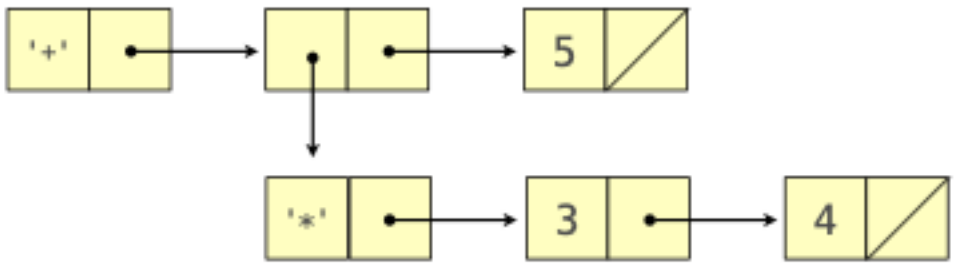
A Scheme call expression is a Scheme list that is represented using a `Link` instance in Python.

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)

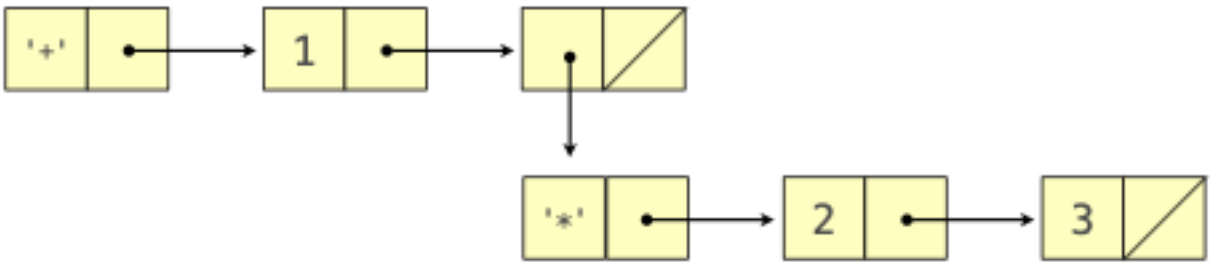
The `Link` class and `nil` object are defined in link.py of the Scheme project.

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

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

nil = Link.empty
```

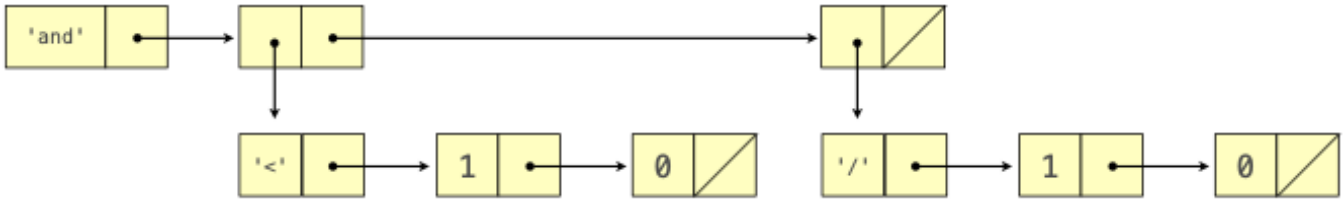
Q1: Representing Expressions

Write the Scheme expression in Scheme syntax represented by each `Link` below.

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




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



Evaluation

Q2: Evaluation

(*Note*: Some past exams have had a question in exactly this format.) 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`
- `-`
- `(`
- `)`

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

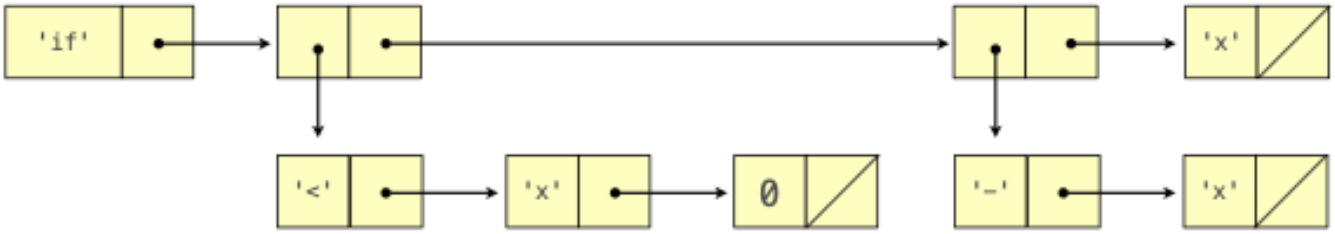
1. `(+ (* 3 4) 5)`
2. `+`
3. `(* 3 4)`
4. `*`
5. `3`
6. `4`
7. `5`

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

Q3: 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):
        "*** YOUR CODE HERE ***"


    else:
        "*** YOUR CODE HERE ***"
```

More Scheme!

Q4: 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 ___________________ ______________________) nil)
        ((= i 0) ______________________________________)
        (else ________________________________________)))
    (slicer-helper lst a b))
  slicer)
```

Q5: Reverse

Write the procedure `reverse`, which takes in a list `lst` and outputs a reversed list.

Hint: you may find the built-in append procedure useful.

```
(define (reverse lst)
    'YOUR-CODE-HERE



)
```