

Switch to Pensieve:

- **Everyone:** Go to pensieve.co, log in with your @berkeley.edu email, and **enter your group number** as the room number (which was in the email that assigned you to this discussion). As long as you all enter the same number (any number), you'll all be using a shared document.

Once you're on Pensieve, you don't need to return to this page; Pensieve has all the same content (but more features). If for some reason Pensieve doesn't work, return to this page and continue with the discussion.

Attendance

Fill out this [discussion attendance form](#) with the unique number you receive from your TA. As soon as you get your number, fill out the form, selecting *arrival* (not *departure* – that's later).

Getting Started

If there are fewer than 3 people in your group, feel free to merge your group with another group in the room.

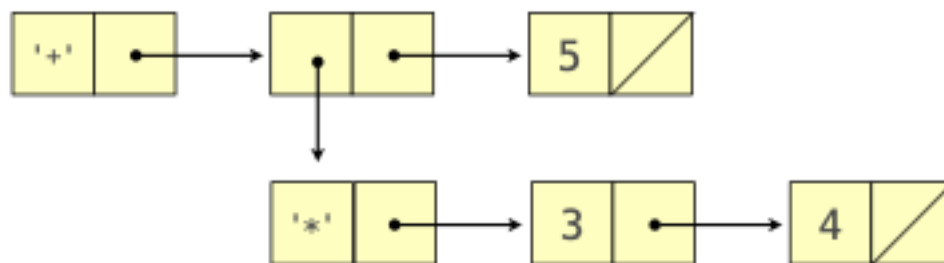
Everybody say your name, and then figure out who is planning to travel outside of the Bay Area the soonest. Feel free to discuss your travel plans.

Representing Lists

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

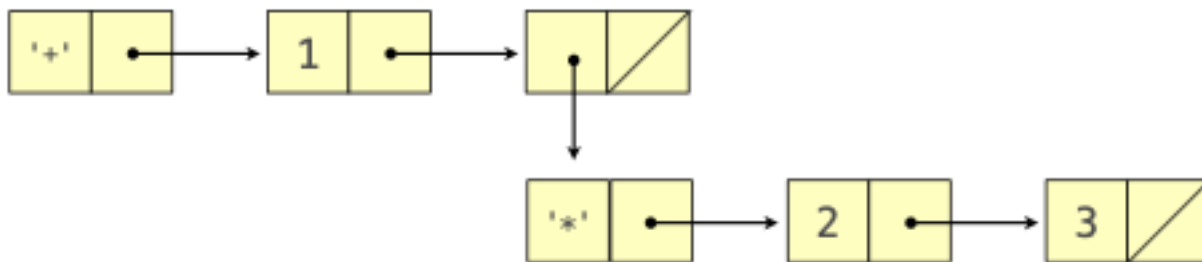
    ... # There are also __str__, __repr__, and map methods, omitted here.
```

Q1: Representing Expressions

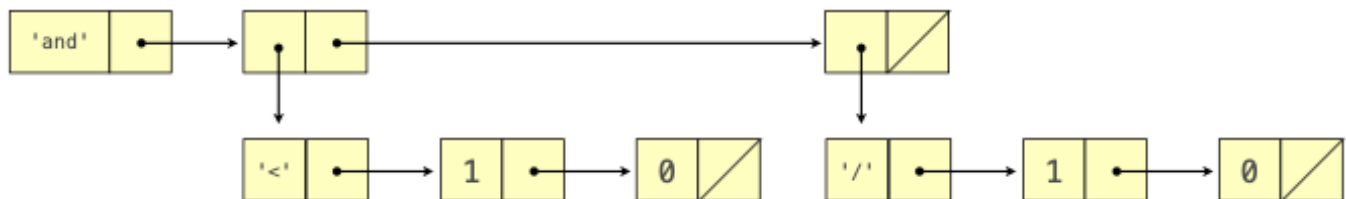
Write the Scheme expression in Scheme syntax represented by each `Pair` below. Try drawing the linked list diagram too. The first one is done for you.

```
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)))
```



Discussion Time: What does `(and (< 1 0) (/ 1 0))` evaluate to? Discuss among your group until you all agree.

#f

Evaluation

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

Discussion time: Describe to each other why * is evaluated and what it evaluates to.

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

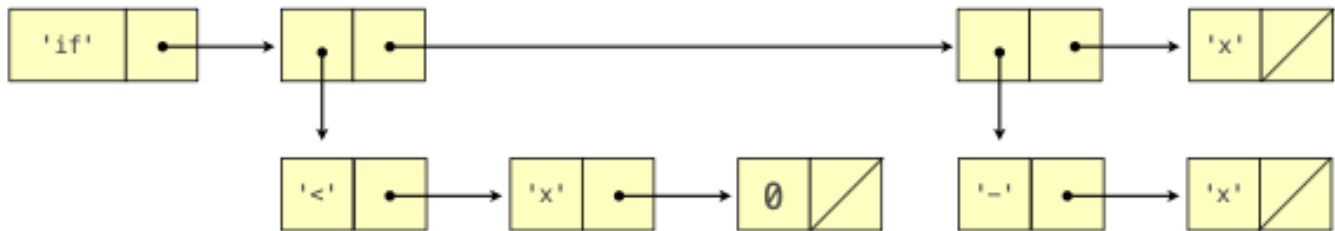
By default, * evaluates to a procedure that multiplies its arguments together. But * could be redefined at any time, and so the symbol * must be evaluated each time it is used in order to look up its current value.

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

An **if** expression is also a Scheme list represented using a Pair instance.

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

```
Pair('if', Pair(Pair('<', Pair('x', Pair(0, nil))), Pair(Pair('-', Pair('x', nil)), Pair('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) 1. (< x 0) 1. < 1. x 1. 0 1. x

Discussion time: Come up with a short explanation of why neither **if** nor - are evaluated even though they both appear in (if (< x 0) (- x) x).

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
- -
- (
-)

With `x` bound to `-2`, `(< x 0)` evaluates to `#t`, and so `(- x)` will be evaluated, but `(if (= x -2) 100 y)` will not. The operator and operands of a call expression are evaluated for every call expression that is evaluated. `(< x 0)` and `(- x)` are both call expressions.

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 `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):
        """ YOUR CODE HERE """

    else:
        """ YOUR CODE HERE """

```

If `expr` is not a pair, then it is a number or '+' or '*'. In all of these cases, the `expr` should be printed to indicate that it would be evaluated.

If `expr` is a pair, then it is a call expression. Print it. Then, the operator and operands are evaluated. These are the elements in the list `expr`. So, iterate through `expr` (using either a `while` statement or `expr.map(...)`) and call `print_evals` on each element.

Document the Occasion

Let your TA know you're done so that you can each get a **departure** number, and fill out the [attendance form](#) again (this time selecting *departure* instead of *arrival*). If your TA isn't in the room, go find them next door.