# Lab 11 Solutions **lab11.zip (lab11.zip)**

## Solution Files

## Introduction

In the Scheme project (https://cs61a.org/proj/scheme/), you'll be implementing a Python interpreter for Scheme.

Part of the process of interpreting Scheme expressions is being able to **parse** a string of Scheme code as our input into our interpreter's internal Python representation of Scheme expressions. As all Scheme expressions are Scheme lists (and therefore linked lists), we represent all Scheme expressions using the `Pair` class, which behaves as a linked list. **This class is defined in `pair.py`.**

When given an input such as `(+ 1 2)`, there are two main steps we want to take.

The first part of interpreting expressions is taking the input and breaking it down into each component. In our example, we want to treat each of `(`, `+`, `1`, `2`, and `)` as a separate token that we can then figure out how to represent. This is called **lexical analysis**, and has been implemented for you in the `tokenize_lines` function in `scheme_tokens.py`.

Now that we've broken down the input into its component parts, we want to turn these Scheme tokens into our interpreter's internal representations of them. This is called **syntactic analysis**, which happens in `scheme_reader.py` in the `scheme_read` and `read_tail` functions.

- `(` tells us we are starting a call expression.
- `+` will be the operator, as it's the first element in the call expression.
- `1` is our first operand.
- `2` is our second operand.
- `)` tells us that we are ending the call expression.

The main idea is that we'd like to first recognize what the input represents, before we do any of the evaluating, or calling the operator on the operands, and so on.

The goal of this lab is to work with the various parts that go into parsing; while in this lab and in the project, we're focusing on the Scheme language, the general ideas of how we're setting up the Scheme interpreter can be applicable to other languages -- such as Python itself!

# Required Questions

Check out the introduction for the context of this lab.

## Part 1

### Context

We store tokens ready to be parsed in `Buffer` instances. For example, a buffer containing the input `(+ (2 3))` would have the tokens `'('` , `'+'` , `'('` , `2` , `3` , `')'` , and `')'` .

In this part, we will implement the `Buffer` class.

A `Buffer` provides a way of accessing a sequence of tokens across lines.

Its constructor takes an iterator, called "the `source` ", that returns the next line of tokens as a list each time it is queried, until it runs out of lines.

For example, `source` could be defined as shown:

```
line1 = ['(', '+', 6, 1 ')']      # (+ 6 1)
line2 = ['(', 'quote', 'A', ')']  # (quote A)
line3 = [2, 1, 0]                 # 2 1 0
input_lines = [line1, line2, line3]
source = iter(input_lines)
```

The `Buffer` in effect concatenates the sequences returned from its source and then supplies the items from them one at a time through its `pop_first` method, calling the `source` for more sequences of items only when needed.

In addition, `Buffer` provides a `current` method to look at the next item to be supplied, without sequencing past it.

### Problem 1

**Important:** Your code for this part should go in `buffer.py`.

Your job in this part is to implement the `current` and `pop_first` methods of the `Buffer` class.

`current` should return the current token of the current line we're on in the `Buffer` instance *without* removing it. If there are no more tokens in the current line, then `current` should move onto the next valid line, and return the **first** token of *this* line. If there are no more tokens left to return from the entire source (we've reached the end of all input lines), then `current` should return `None` (this logic is already provided for you in the `except StopIteration` block).

If we call `current` multiple times in a row, we should get the same result since calls to `current` won't change what token we're returning.

You may find `self.index` helpful while implementing these functions, but you are not required to reference it in your solution.

**Hint:** What instance attribute can we use to keep track of where we are in the current line?

**Hint:** If we've reached the end of the current line, then `self.more_on_line()` will return `False`. In that case, how do we "reset" our position to the beginning of the next line?

`pop_first` should return the current token of the `Buffer` instance, and move onto the next potential token (to be returned on the next call to `pop_first`). If there are no more tokens left to return from the entire source (we've reached the end of all input lines), then `pop_first` should return `None`.

**Hint:** Do we need to update anything to move onto the next potential token?

Use Ok to test your code:

```
python3 ok -q buffer
```

## Part 2

## Internal Representations

The reader will parse Scheme code into Python values with the following representations:

| **Input Example** | **Scheme Expression Type** | **Our Internal Representation** |
|---|---|---|
| `scm> 1` | Numbers | Python's built-in `int` and `float` values |
| `scm> x` | Symbols | Python's built-in `string` values |
| `scm> #t` | Booleans ( `#t` , `#f` ) | Python's built-in `True` , `False` values |
| `scm> (+ 2 3)` | Combinations | Instances of the `Pair` class, defined in `scheme_reader.py` |
| `scm> nil` | `nil` | The `nil` object, defined in `scheme_reader.py` |

When we refer to combinations here, we are referring to both call expressions and special forms.

## Problem 2

**Important:** Your code for this part should go in `scheme_reader.py` .

Your job in this part is to write the parsing functionality, which consists of two mutually recursive functions: `scheme_read` and `read_tail` . Each function takes in a single `src` parameter, which is a `Buffer` instance.

- `scheme_read` removes enough tokens from `src` to form a single expression and returns that expression in the correct internal representation.
- `read_tail` expects to read the rest of a list or `Pair` , assuming the open parenthesis of that list or `Pair` has already been removed by `scheme_read` . It will read expressions (and thus remove tokens) until the matching closing parenthesis `)` is seen. This list of expressions is returned as a linked list of `Pair` instances.

In short, `scheme_read` returns the next single complete expression in the buffer and `read_tail` returns the rest of a list or `Pair` in the buffer. Both functions mutate the buffer, removing the tokens that have already been processed.

The behavior of both functions depends on the first token currently in `src` . They should be implemented as follows:

`scheme_read` :

- If the current token is the string `"nil"` , return the `nil` object.

- If the current token is `(`, the expression is a pair or list. Call `read_tail` on the rest of `src` and return its result.
- If the current token is `'`, the rest of the buffer should be processed as a `quote` expression. You will implement this portion in the next problem.
- If the next token is not a delimiter, then it must be a primitive expression (i.e. a number, boolean). Return it. **Provided**
- If none of the above cases apply, raise an error. **Provided**

`read_tail`:

- If there are no more tokens, then the list is missing a close parenthesis and we should raise an error. **Provided**
- If the token is `)`, then we've reached the end of the list or pair. **Remove this token from the buffer** and return the `nil` object.
- If none of the above cases apply, the next token is the operator in a combination. For example, `src` could contain `+ 2 3)`. To parse this:
  1. `scheme_read` the next complete expression in the buffer.
  2. Call `read_tail` to read the rest of the combination until the matching closing parenthesis.
  3. Return the results as a `Pair` instance, where the first element is the next complete expression from (1) and the second element is the rest of the combination from (2).

Use Ok to unlock and test your code:

```
python3 ok -q scheme_read -u
python3 ok -q scheme_read
```

## Problem 3

**Important:** Your code for this part should go in `scheme_reader.py`.

Your task in this problem is to complete the implementation of `scheme_read` by allowing the function to now be able to handle quoted expressions.

In Scheme, quoted expressions such as `'<expr>` are equivalent to `(quote <expr>)`. That means that we need to wrap the expression following `'` (which you can get by recursively calling `scheme_read`) into the `quote` special form, which is a Scheme list (as with all special forms).

In our representation, a `Pair` represents a Scheme list. You should therefore wrap the expression following `'` in a `Pair`.

For example, `'bagel`, or `["'", "bagel"]` after being tokenized, should be represented as `Pair('quote', Pair('bagel', nil))`. `'(1 2)` (or `["'", "(", 1, 2, ")"]`) should be represented as `Pair('quote', Pair(Pair(1, Pair(2, nil)), nil))`.

Use Ok to unlock and test your code:

```
python3 ok -q quote -u
python3 ok -q quote
```

## Running your parser

Now that your parser is complete, you can test the read-eval-print loop by running:

```
python3 scheme_reader.py --repl
```

Every time you type in a value into the prompt, both the `str` and `repr` values of the parsed expression are printed. You can try the following inputs:

```
read> 42
str : 42
repr: 42
read> nil
str : ()
repr: nil
read> (1 (2 3) (4 (5)))
str : (1 (2 3) (4 (5)))
repr: Pair(1, Pair(Pair(2, Pair(3, nil)), Pair(Pair(4, Pair(Pair(5, nil), nil)), nil)))
```

To exit the interpreter, you can type `exit`.

## Submit

Make sure to submit this assignment by running:

```
python3 ok --submit
```