carlae : Part II

Due Dates:

- *Submission to website:* Monday, May 1, 10pm
- *Checkoff by LA/TA*: Thursday, May 4, 10pm

This lab assumes you have Python 3.5 or newer installed on your machine.

It is alright to use modules that are already imported for you in the template, but you should not add any imports to the code skeleton.

Introduction

In lab 8A, you implemented an interpreter for a dialect of LISP called *carlae*. This lab builds on your work from lab 8A to introduce some new features into the *carlae* language. If you have not yet finished lab 8A, you should do so before working on this lab.

Start by copying your `lab.py` from lab 8A into this week's code distribution.

Lists

Our first new piece in this lab will be adding support for *lists*. Add a new function called `list` to the built-in functions. `list` should take arbitrarily many arguments, and it should return a list containing those elements. In *carlae*, we will implement lists as linked lists (as is typical in many LISP dialects). The implementation is discussed below:

Each instance of `LinkedList` should have only two instance variables:

- an attribute called `elt` (the element at this position in the list), and
- an attribute called `next`, which points to an instance of `LinkedList` representing the next element in the list, or `None` if this instance represents the end of the list.

You should make sure that calling `list` with no arguments produces a representation for an empty list (you may take some inspiration from the fact that the `next` attribute of a length-1 list is `None`).

Built-in List Functions

Add a `list` function to *carlae*; this function should take one or more arguments and should construct a `LinkedList` instance that contains those arguments, in order. In addition, we will define some additional built-in functions for operating on lists within *carlae*:

`car` and `cdr`

These functions, named for historical reasons, return the "head" and the "tail" of a linked list, respectively.

- `(car LIST)` should take a list as argument and should return the first element in that list.
- `(cdr LIST)` should take a list as argument and return the list containing all but the first element in that list.

Convenience Methods

In addition, we will add a few functions for convenience:

- `(length LIST)` should take a list as argument and should return the length of that list.
- `(elt-at-index LIST INDEX)` should take a list and an index, and it should return the element at the given index in the given list (as in Python, indices start from 0).
- `(concat LIST1 LIST2 LIST3 ...)` should take an arbitrary number of lists as arguments and should return a new list representing the concatenation of these lists. If exactly one list is passed in, it should return a copy of that list. If no `concat` is called with no arguments, it should produce an empty list.

`map`, `filter`, and `reduce`

Beyond these functions, the following will allow us to easily construct new lists from existing ones:

- `(map FUNCTION LIST)` takes a function and a list as arguments, and it returns a *new list* containing the results of applying the given function to each element of the given list.

  For example, `(map (lambda (x) (* 2 x)) (list 1 2 3))` should produce the list `(2 4 6)`.

- `(filter FUNCTION LIST)` takes a function and a list as arguments, and it returns a *new list* containing only the elements of the given list for which the given function returns true.

  For example, `(filter (lambda (x) (> x 0)) (list -1 2 -3 4))` should produce the list `(2 4)`.

- `(reduce FUNCTION LIST INITVAL)` takes a function, a list, and an initial value as inputs. It produces its output by successively applying the given function to the elements in the list, maintaining an intermediate result along the way. This is perhaps the most difficult of the three functions to understand, but it is perhaps easiest to see by example.

  Consider `(reduce * (list 9 8 7) 1)`. The function in question is `*`. Our initial value is `1`. We take this value and combine it with the first element in the list using the given function, giving us `(* 1 9)` or `9`. Then we take *this* result and combine it with the next element in the list using the given function, giving us `(* 9 8)` or `72`. Then we take *this* result and combine it with the next element in the list using the given function, giving us `(* 72 7)` or `504`. Since we have reached the end of the list, this is our final return value (if there were more elements in the list, we would keep combining our "result so far" with the next element in the list, using the given function).

The Wikipedia pages for map, filter, and reduce provide some additional examples.

Once we have these three functions, we have the equivalent of list comprehensions in *carlae*! In Python, for example, we might write:

```
sum([i**2 for i in some_list if i < 0)
```

In *carlae*, we can now do the same thing with the following code:

```
(reduce + (map (lambda (i) (* i i)) (filter (lambda (i) (< i 0)) some_list)) 0)
```

This is a lot to take in, but it gives us the same result:

- It first *filters* `some_list` to produce a list containing only the negative values.
- It then *maps* the "square" function onto the resulting list.
- Finally, it *reduces* that result by successive application of the `+` operator to produce the sum.

Implementation

Implement the `list`, `car`, `cdr`, `length`, `elt-at-index`, `concat`, `map`, `filter`, and `reduce` functions and add them to the built-in functions. Once you have done so, your code should pass tests 1-19 in `test.py`.

Reading From Files

So far, when we have wanted to define new functions, we have had to type them (or copy/paste them) into the REPL every time we run the program. In this section, we will add the capability to run the contents of a file before dropping into our REPL, which we can use, for example, to define multiple functions.

Define a function called `evaluate_file` in `lab.py`. This function should take a single argument (a string containing the name of a file to be evaluated) and an optional argument (the environment in which to evaluate the expression), and it should return the result of evaluating the expression contained in the file.

You may find the documentation for Python's built-in `open` function helpful.

After implementing `evaluate_file`, your code should pass tests 1-22 in `test.py`.

Command Line Arguments

Now that we have the ability to evaluate the contents of a file in a particular environment, we will need to let Python know *which* files it should evaluate before dropping into the REPL. We will do this by passing the names of these files to *carlae* as *command line arguments*. For example, instead of just running:

```
$ python3 lab.py
```

we will run something like:

```
$ python3 lab.py some_definitions.crl more_definitions.crl
```

From inside of Python, these arguments are available as part of the `sys.argv` list. For the example above, `sys.argv` will contain:

```
['lab.py', 'some_definitions.crl', 'more_definitions.crl']
```

Modify `lab.py` so that, when `lab.py` is run with filenames passed in on the command line, it evaluates the expressions contained in those files into the global environment before entering the REPL. You may assume that each file contains only one expression. To test your code, you can make a few files that contain simple expressions you can check (for example, `define` a single variable inside a file and make sure it is available to you from the REPL after that file is evaluated).

Evaluating Multiple Expressions with `begin`

To help with the above, introduce a new built-in function called `begin`. `begin` should simply return its last argument. This is a useful structure for running commands successively: even though only the last argument is returned, all of the arguments are evaluated in turn, which allows us to run arbitrary expressions sequentially.

After you have implemented `begin`, you should be able to run `python3 lab.py resources/definitions.crl` to grab some definitions into the REPL.

After implementing `begin`, your code should pass tests 1-25 in `test.py`.

`let` and `set!`

We will finish by implementing two additional special forms, which can be used to manipulate variable bindings: `let`, and `set!`.

`let`

`let` is used for creating *local variable definitions*. It takes the form: `(let ((VAR1 VAL1) (VAR2 VAL2) (VAR3 VAL3) ...) BODY)`, where `VAR1`, `VAR2`, etc, are variable names, and `VAL1`, `VAL2`, etc, are expressions denoting the values to which those names should be bound. It works by:

- Evaluating all the given values in the current environment
- *Creating a new environment* whose parent is the current environment, and binding each name to its associated value in this new environment.
- Evaluating the `BODY` expression in this new environment (this value is the result of evaluating the `let` special form).

Note that the given bindings are only available in the body of the `let` expression. For example:

```
in> (define z 5)
    out> 5

in> (let ((x 5) (y 3)) (+ x y z))
    out> 13

in> x
    EXCEPTION!

in> y
    EXCEPTION!

in> z
    out> 5
```

`set!`

`set!` (often pronounced "set-bang", or just "set") is used for *changing the value of an existing variable*. It takes the form: `(set! VAR EXPR)`, where `VAR` is a variable name, and `EXPR` is an expression. It should work by finding the first environment in which `VAR` is defined, and updating its binding *in that environment* to be the result of evaluating `EXPR`, and it should also evaluate to that same value. If `VAR` is not defined in any environments in the chain, `set!` should raise an `EvaluationException`.

```
in> (define x 7)
    out> 7

in> (define (foo z) (set! x (+ z 2)))
    out> function object

in> (foo 3)
    out> 5

in> x
    out> 5

in> (define (bar z) (define x (+ z 2)))
    out> function object

in> (bar 7)
    out> 9

in> x
    out> 5
```

Implement `let` and `set!` in `lab.py`. After doing so, your code should pass all 33 tests in `test.py`!

Coding Points

There are 20 coding points in this lab:

- 5 will be awarded for "style," based on the structure and clarity of your code, and accompanying comments/docstrings.
- 10 will be awarded for your implementation of `LinkedList`.
- 5 will be awarded for correctly implementing the REPL, including reading in files from the command line.

Final Notes

Over the last two weeks, we have developed an interpreter for a "little" language called *carlae*. We started out small, but by making incremental changes, we've worked our way to an interpreter that is capable of evaluating arbitrarily-complicated programs! And what's more, we added some additional features to make interactions with the language more convenient.

What follows is a few suggestions of neat (but optional) additional features that you might consider adding to your interpreter. Although these are not required, they would make great extra practice if you're looking for it!

First Optional Improvement: Tail-Call Optimization

If you have the time and interest, a *really* interesting and powerful optimization for our interpreter comes in the form of *tail-call optimization* (other suggestions are available on the following page).

A typical way of structuring the evaluator from above involves making recursive calls to `evaluate`, in particular when calling functions. This is a nice way of structuring things, but it actually leads to issues. For example, try calling `(fib 2000)` from your REPL after loading in the definitions from `resources/definitions.crl`. What happens?

We run into issues with recursive calls because Python (necessarily) has a limit on the "depth" it will allow in a recursive call, to prevent infinite recursion. Even if Python didn't have this limit, it would end up using quite a lot of memory allocating new stack frames for these recursive calls.

A neat optimization to avoid this problem is to implement tail-call optimization, whereby we can avoid some of these issues (allowing, for example, computing `(fib n)` for arbitrarily large `n`).

In short, many pieces of our interpreter involve returning the result of a single new call to `evaluate`, with a different expression and a different environment. In those situations (conditionals, calling user-defined functions, etc), it would be much better from an efficiency perspective to avoid the recursive call to `evaluate`; rather, we can simply adjust the values of the expression and the environment within the same call to `evaluate` by introducing a looping structure: keep looping until we have successfully evaluated a structure, and if the result is simply the result of evaluating another expression (potentially in a different environment), then adjust those values as necessary.

There are no tests for this optimization, but after doing so, your code should work for `(fib 100000)` (or arbitrarily-high `n`)!

Additional (Optional) Improvements and Exercises

If you have the time and interest, you can improve upon this base language by implementing some additional features. Below are some ideas for possible improvements, or just for ways to get extra practice. These are by no means required, but we are still happy to help with them if you get stuck!

- Add support for mutating lists by implementing functions `set-car` and `set-cdr` to modify the `elt` and `next` attributed of instances of `LinkedList`. Then try using these to implement a `set-elt-at-index` function in *carlae*.
- Add support for the `quote` and `unquote` special forms (search for "special form: quote" on this page).
- Implement strings as an additional data type (be careful of how you handle comments to make sure that a `;` within a string doesn't get treated as the start of a comment!).
- Allow the body of a function defined with `lambda`, or the body of a `let` expression, to consist of more than one expression (this should be implicitly translated to a `begin` expression).
- Improve the system's error reporting by providing meaningful error messages that describe what error occurred.
- Since *iteration* doesn't exist in *carlae* (except via recursion), try implementing some simple programs in *carlae* for extra practice with recursion!

- Add support for importing functions and constants from Python modules (by mapping *carlae* functions to the the `__import__` and `getattr` Python functions) so that the following will work:

```
in> (define math (py-import (quote math)))
in> (define sqrt (getattr math (quote sqrt)))
in> (sqrt 2)
    out> 1.4142135623730951
```