Lecture 29: Project 4 Overview

General Comments

- This project is about *reading* programs as well as writing them. Don't just treat the framework you're given as a bunch of magic incantations. Try to understand and learn from it.
- Don't allow yourself to get lost. Keep asking about things you don't understand until you do understand.
- You are always free to introduce auxiliary functions to help implement something. You *do not* have to restrict your changes to the specifically marked areas.
- You are also free to modify the framework outside of the indicated areas in any other way you want, as long as you meet the requirements of the project.
  - Feel free to add new Turtle methods to `scheme_primitives.py` or new standard functions to `scheme_prelude.scm`.
  - Feel free to refactor code.
  - *ALWAYS* feel free to fix bugs in the framework (and tell us!).
- Stay in touch with your partner! If you're having problems getting along, tell us early, or we probably won't be able to help.

Interpreting Scheme

- Your project will have a structure similar to the calculator:
  - Split input into tokens.
  - Parse the tokens into Scheme expressions.
  - Evaluate the expressions.
- Evaluation breaks into cases:
  - Numerals and booleans evaluate to themselves.
  - Symbols are evaluated in the current environment (needs a data structure).
  - Combinations are either
    - Special forms (like `define` or `if`), each of which is a special case, or
    - Function calls

Major Pieces

- `read_eval_print` is the main loop of the program, which takes over after initialization. It simply reads Scheme expressions from an input source, evaluates them, and (if required) prints them, catching errors and repeating the process until the input source is exhausted.
- `tokenize_lines` in `scheme_tokens.py` turns streams of characters into tokens. You don't have to write it, but you should understand it.
- The function `scm_read` parses streams of tokens into Scheme expressions. It's a very simple example of a *recursive-descent parser.*
- The class `EnvironFrame` embodies environment frames. You fill in the method that creates local environments.
- The class `Evaluation` embodies the process of evaluating an expression. Understand how it *all* works and fill in the missing bits.
- `scheme_primitives.py` defines the basic Scheme expression data structure (aside from functions) and implements the "native" methods (those implemented directly in the host language: Python, or in other compilers, C).

Function Calls

- The idea here is a "mutually recursive dance" between two parties (just like the calculator):
  - `eval`, which evaluates operator and operands, and
  - `apply`, which applies functions to the resulting values.
- Interestingly, these just happen to be standard functions in the language we are defining: we could in principle (and fact) interpret Scheme in Scheme *metacircularly*.
- But if we want to do this in Python, we have to deal with proper tail recursion (i.e., its lack in Python vs. its presence in Scheme).
- That is, a purely tail-recursive function must be able to run arbitrarily long (without overflowing any internal stack).

Dealing With Tail Recursion

- To handle tail recursion, you'll actually implement a slightly modified version of `eval`, one which *partially evaluates* its argument, performing one "evaluation step."
- Each evaluation step returns *either* a value (in which case, evaluation of the expression is done), or another expression and the environment in which to evaluate it.
- So now Python can iterate:

```
def eval(expr, environ):
    while expr is still an unevaluated expression,
          expr, environ = eval_step(expr, expression)
    return expr
```

Example

- Consider problem of getting kth item in list:

```
;; Element #K of L
(define (list-ref L k)
    (if (= k 0) (car L) (list-ref (cdr L) (- k 1))))
```

- We want to evaluate `(list-ref '(3 5 7) 2)`.
- Let's represent the state of an evaluation as a stack of *"evaluation frames"* (class `Evaluation`), each of which looks like this when partially evaluated:

Expression	Value	Environment
(list-ref (cdr L) (- n 1))		L: (3 5 7), k: 2, *globals*

or like this when fully evaluated:

	7	L: (7), k: 0, *globals*

Example: list-ref

```
(define (list-ref L k)
    (if (= k 0) (car L) (list-ref (cdr L) (- k 1))))
```

First, the call:

Expression	Value	Environment
(list-ref '(1 2 3) 2)		*globals*

After evaluating the quoted expression, we *replace* the call with the body:

Expression	Value	Environment
(if . . .)		L: (3 5 7), k: 2, *globals*

Now evaluate the condition (recursively, in another `Evaluation`):

Expression	Value	Environment
(= k 0)		L: (3 5 7), k: 2, *globals*
(if . . .)		L: (3 5 7), k: 2, *globals*

Example (contd.)

Expression	Value	Environment
(= k 0)		L: (3 5 7), k: 2, *globals*
(if . . .)		L: (3 5 7), k: 2, *globals*

Evaluate the primitive function call =:

Expression	Value	Environment
	#f	L: (3 5 7), k: 2, *globals*
(if . . .)		L: (3 5 7), k: 2, *globals*

Which causes us to replace the `if` with its "false" branch:

Expression	Value	Environment
(list-ref (cdr L) (- k 1))))		L: (3 5 7), k: 2, *globals*

Example (contd.)

Expression	Value	Environment
(list-ref (cdr L) (- k 1))))		L: (3 5 7), k: 2, *globals*

After evaluating `list-ref` (to get a function), `(cdr L)`, and `(- k 1)` (recursively, each in its own `Evaluation`), we *replace* the call on `list-ref` with the body:

Expression	Value	Environment
(if ...)		L: (5 7), k: 1, *globals*

and so on. Thus, the stack of evaluations-in-progress does not keep growing.

Application and evaluation in Project 4

- In the class `Evaluation`, you'll find a method `step`, which carries out one step in the process described above: it either
  - Computes the final value of an expression (if it is a symbol, quote, literal, or call on a primitive function), or
  - Partially evaluates the expression, and then replaces it with a new expression and environment that carries out the rest of the computation.
- In the subclasses `LambdaFunction` and `PrimitiveFunction`, you'll find the `apply_step` method, which operates on `Evaluations` and is how the `step` method deals with function calls once it has values for the operator and operands. It either:
  - Calls the indicated primitive function and gives the `Evaluation` its final value (`PrimitiveFunction`), or
  - Replaces the `Evaluation`'s expression and environment with the body and environment of its function (a `LambdaFunction`).

Handling Special Forms

- The "special" forms (expressions that don't obey the usual evaluate-all-operands-and-call rule) all get handled by eponymous methods in `Evaluation` (e.g., `do_cond_form`).
- Unlike `apply_step`, they exert explicit control over their operand's evaluation.
- Some special forms can be rewritten into equivalent Scheme expressions that replace the original, but this is up to the implementor.
- In fact, full Scheme has *macros*, which are Scheme functions that produce Scheme expressions. To evaluate a macro call, an interpreter:
  - Calls the macro function without evaluating and operands ("quotes the operands").
  - Then evaluates the expression that is returned.
- It is a powerful, but often messy, feature.