Lecture 30: Mostly Project 4 Overview

Stream Example

```
f = Stream(1,
           lambda: Stream(1,
                          lambda: combine_streams(add, f, f.rest)))
```

- This creates a new stream that initially contains only one item and a lazy continuation:

  ```
  f = [ 1, lambda: Stream(1, lambda: combine_streams(add, f, f.rest))) ]
  ```

- If I ask for `f.rest`, I get

  ```
  f = [ 1, 1, lambda: combine_streams(add, f, f.rest))) ]
  ```

- And now `f.rest.rest`:

  ```
  f = [ 1, 1, 2, lambda: combine_streams(add, f.rest, f.rest.rest))) ]
  ```

- And now `f.rest.rest.rest`:

  ```
  f = [ 1, 1, 2, 3,
        lambda: combine_streams(add, f.rest.rest, f.rest.rest.rest))) ]
  ```

- Do you see where this is going?

Project 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 via email!).
- 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_loop` is the main loop of the program, which takes over after initialization. Reads Scheme expressions from an input source, evaluates (interprets) them, and prints the result, 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 `scheme_read` parses streams of tokens into Scheme expressions. It's a very simple example of a *recursive-descent parser.*
- The class `Frame` embodies environment frames. You fill in the method that creates local environments.
- The `scheme_eval` function evaluates a Scheme 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).

Scheme Values

- The project uses an object-oriented approach in its internal representation of values.
- The class `SchemeValue` is at the top of the hierarchy. Defines default definitions of all methods.
- `SchemeValue` is (in effect) *abstract:* there are no objects whose type is just plain `SchemeValue`.
- Instead, all Scheme values are represented by subtypes of `SchemeValue`.

Type Hierarchy

- SchemeValue
  - SchemeNumber
    - SchemeInt: 3
    - SchemeFloat: 2.5
  - SchemeSymbol: foo (value of 'foo)
  - SchemeStr: "bar"
  - scheme_true, scheme_false: #t, #f
  - Pair: (1 . 2)
  - nil: ()
  - Procedure
    - PrimitiveProcedure: value of +
    - LambdaProcedure: value of (lambda (x) x)
    - MuProceure (extension)
    - NuProcedure (extra-credit extension)
  - okay: the "undefined value"

Example: Implementing apply

- Obvious way to handle, e.g., function application:

```
def scheme_apply(procedure, args, env):
    if isinstance(procedure, PrimitiveProcedure):
       Apply primitive procedure
    elif isinstance(procedure, LambdaProcedure):
       Apply lambda procedure
    ...
    else:
       raise SchemeError("not a procedure")
```

- That is, all code about function application is collected in one place, with a bunch of conditionals to figure out which to use.
- Adding a new type of procedure means also modifying this function and any other such collective function on procedure values.

Example: The Object-Oriented Approach

- Our project uses a different approach, typical of object-oriented designs:
- `scheme_apply` calls `procedure.apply(args, env)`, a method in all `SchemeValues`.
- Then, there are definitions for each type that supports function application:

```
class SchemeValue:
    def apply(self, args, env): raise SchemeError("not a procedure")
    ...
class Procedure(SchemeValue):
    ...
class PrimitiveProcedure(Procedure):
    def apply(self, args, env): Apply primitive procedure
    ...
class LambdaProcedure(Procedure):
    def apply(self, args, env): Apply lambda procedure
    ...
```

- Other classes, such as `SchemeInt`, inherit the default (error) definition from `SchemeValue`.

The scheme_eval function

- Could use an object-oriented approach for `scheme_eval`, the function that interprets Scheme values as Scheme programs.
- But representing Scheme programs is just one of many uses.
- Makes less sense to build evaluation functions into, e.g., class `Pair`.
- So our `scheme_eval` uses the alternative approach of explicit type or value testing to decide what to do.
- For example (simplified from the project):

```
def scheme_eval(expr, env):
    ...
    if scheme_symbolp(expr):
        return env.lookup(expr)
    elif scheme_atomp(expr):
        return expr

    elif not scheme_listp(expr):
        raise SchemeError("malformed list")
    else:  Handle combinations
```

Special Forms

- The Scheme special forms (e.g., `(lambda ...)`, `(if ...)`) are those that are not treated as plain function calls (evaluate arguments, apply function).
- They all take the same arguments—the rest of the form and the environment—and all return the same thing: a new expression and a new environment.
- So we use a dispatch table to handle them:

```
if (scheme_symbolp(first) and first in SPECIAL_FORMS):
    ...
    expr, env = SPECIAL_FORMS[first](expr.second, env)
    ...
```

Nu Procedures

- One of the extra-credit problems has you introduce *call-by-name* parameter passing.
- Idea is to have a type of procedure that evaluates its arguments *lazily*—only when their value is actually needed.
- The term for the usual evaluate-arguments-and-apply semantics is *applicative-order evaluation*.
- This lazy evaluation is known as *normal-order evaluation.*
- For example, here's an example you've seen in HW#1 in another form:

```
(define if-function
   (nu (test true-part false-part) (if test true-part false-part)))
```

- If `if-function` were an ordinary (lambda) function, then the call `(if-function (zero?  x) 1 (/ 1 x))` would fail if `x` is 0.
- But as a call-by-name function, the division is never executed in that case.

Tail recursion

- You'll see that `scheme_eval` is a bit more complicated than what we've talked about.
- It contains a loop so that a single call might involve several function applications or expression evaluations.
- However, to start with, things are arranged "classically":
  - For a call, `scheme_eval` evaluates the operands recursively, applies the resulting procedure to the resulting argument values, and returns the result.
- For the first extra-credit problem, you'll be asked to modify `scheme_eval` to instead loop instead of recurse when it is possible to do so correctly.
- Hint: this is a very easy extra-credit problem!