Lecture 27: Anatomy of an Interpreter

Interpreting Scheme

- Your project will have a structure similar to the calculator:
  - Split input into *tokens*, which are themselves *Scheme values*.
  - *Parse* the tokens into Scheme *expressions*, which are also Scheme values.
  - *Evaluate* the expressions to produce Scheme values.

Evaluation

Evaluation breaks into cases:

- Numerals, strings, booleans, and the empty list "evaluate to themselves" (are "*self-evaluating*"):

```
3 ===> 3        () ===> ()     #t ===> #t    "hello" ===> "hello"
```

  [Here, E `===>` V means "expression E evaluates to value V"]

- Symbols are evaluated in the *current environment* (a Python data structure that is internal to the interpreter).

```
(define x 3) ;; Sets x to 3 in the current environment
x ===> 3
```

- Combinations (represented by Scheme pairs) are either
  - Special forms (like `define` or `if`), each of which is a special case:

```
(if (> 3 2) 1 (/ 2 0)) ===> 1
```

  - or Function calls

```
(+ 1 2) ===> 3
```

Metacircularity: Scheme in Scheme

- Tokens, expressions, and the results of evaluating them are all Scheme values.
- We could represent environment frames as well with a Scheme data structure (an association list, as described in Lecture #26).
- At that point, it becomes evident we could easily write this Scheme interpreter in Scheme!
- As we'll see, an observation very much like this led to a major earthquake in 20th century mathematics—one that you'll actually be able to understand just from the content of this course!

Major Pieces

- `read_eval_print_loop` 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.
- `scm_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.
- `scheme_eval` evaluates an expression.
- `scheme_apply` Evaluates a call on a function once the arguments are evaluated.
- Basic (primitive) functions are implemented directly by the interpreter (in Python).

Function Calls

- The idea here is a "mutually recursive dance" between two parties:
  - `scheme_eval`, which evaluates operator and operands, and
  - `scheme_apply`, which applies functions to the resulting values.
- The intepreter sees the expression

  ```
  (f (+ 1 2) y)
  ```

  (which is Scheme data). What does it do?
- First, it *identifies* this as a call: it's a non-empty list, and its first item is not "special" (like **if**).
- So it (recursively) evaluates `f`, `(+ 1 2)`, and `y`, giving a function and two other values.
- Now it calls the function with those arguments, using `scheme_apply`.
- If `f` is a non-primitive function (defined via `define` in the same program), then `apply` will eventually have to *evaluate* its body...
- ...for which it uses `scheme_eval`.
- And so it goes, back and forth between `scheme_eval` and `scheme_apply` until we get down to primitive functions and other base cases.

Tail Recursion

- Consider

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

- This is a tail-recursive call. According to Scheme semantics, it must work, regardless of the length of `L`—no "stack overflow" allowed.
- But if the interpreter naively calls `scheme_eval` to evaluate the body, which calls `scheme_apply` to make the recursive call, which calls `scheme_eval` to evaluate the body..., there will be trouble!

Dealing With Tail Recursion

- To handle tail recursion, you'll actually implement a slightly modified version of `scheme_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 *replaces* a tail-recursive call and current environment with the the body of the function and that function's environment (and loops).

Example

- Consider again the tail-recursive example:

```
;; 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.

Handling Special Forms

- In your project, the "special" forms (expressions that don't obey the usual evaluate-all-operands-and-call rule) all get handled by eponymous functions (e.g., `do_cond_form`).
- Unlike `scheme_apply`, 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 you can add for extra credit), which are Scheme functions that produce Scheme expressions. To evaluate a macro call, an interpreter:
  - Calls the macro function without evaluating its operands ("quotes the operands"), getting back a Scheme expression.
  - It then evaluates the expression that is returned.
- It is a powerful, but often messy, feature.