Lecture 27: Interpreting Scheme

A Scheme interpreter is essentially an extension of the calculator:

- A component known as the reader (scheme_read) reads Scheme values (atoms and pairs).
- Since Scheme expressions and programs are a subset of Scheme values, no further parsing is necessary.
- A function scheme_eval evaluates Scheme expressions.
  - Atoms are its base cases.
  - For function calls, it uses a function scheme_apply, as for the calculator.

Reading

- The project skeleton defines a class Buffer (in buffer.py), whose purpose is to take sequences of tokens (strings) and concatenate them into a single sequence in which one can either look at and, if desired, remove, one token at a time.
- These sequences of tokens come from a method tokenize_lines which breaks sequences of strings into tokens:
  ```python
  >>> from scheme_tokens import tokenize_lines
  >>> from buffer import Buffer
  >>> L = tokenize_lines(['(define x", " (+ y 3))"'], ['b = Buffer(L)']
  >>> b = Buffer(L)
  >>> b.remove_front()
  '('
  >>> b.remove_front()
  'define'
  ```

Apply

- The interpreter function scheme_apply(func, args) has the effect of allowing one to construct and evaluate function calls.
- It has the essentially the same effect that func(*args) does in Python programs.
- In the interpreter, scheme_apply itself has two cases:
  - Either func is a primitive, built-in function, in which case, its code is part of the interpreter, or
  - func is a user-defined function, in which case its code is stored in it as a Scheme expression, and is evaluated by eval.
- So there is a "recursive dance" back and forth between scheme_eval, and scheme_apply.

scheme_read

- Finally, the function scheme_read, which you will complete, pulls tokens off a Buffer until it has a complete Scheme expression:
  ```python
  >>> from scheme_tokens import tokenize_lines
  >>> from buffer import Buffer
  >>> from scheme_reader import scheme_read
  >>> L = tokenize_lines(['(define x", " (+ y 3))", "(define y 42)"'])
  >>> b = Buffer(L)
  >>> scheme_read(b)
  Pair('define', Pair('x', Pair('+', Pair('y', Pair(3, nil))), nil)))
  >>> scheme_read(b)
  Pair('define', Pair('y', Pair(42, nil)))
  ```
Evaluation for Scheme

- Simple expressions are evaluated as for the calculator.
- A Scheme expression consisting of a number simply evaluates to that number. It is self-evaluating.
- A function call \((E_0 \ E_1 \ \cdots \ E_n)\) is evaluated by recursively evaluating the \(E_i\) and then using scheme_apply.
- But Scheme has a number of other cases to handle.

Aside: accessing scheme_eval and scheme_apply in Scheme

- In full Scheme, the functions scheme_eval and scheme_apply are both available to the programmer in the form of the two built-in functions apply and eval:
  ```scheme
  >>> (define L '(1 2 3))
  >>> (apply + L)
  6
  >>> (eval (list '+ 1 2) (scheme-report-environment 5))
  3
  >>> (eval '(+ 1 2) (scheme-report-environment 5))
  3
  ```
- The second argument here, as for scheme_eval, is an environment defining symbols’ values.
- In official Scheme, however, there is no way to get the current environment (the one containing your own definitions), although various implementations do provide a way.

Evaluation of Symbols

- In Scheme expressions, most symbols represent identifiers, which we did not encounter in the calculator.
- Obviously, we need more information to evaluate a symbol than just the symbol itself.
- Fortunately, we already know what’s needed: an environment.
- Thus, to evaluate a Scheme expression, we will need both the expression itself and the environment in which to evaluate it.
- As it happens, exactly the same kind of structure as in Python—environment frames linked by parent pointers—is what we need to interpret Scheme.
- This is because Scheme uses nearly the same scope rules as Python does.
- Earlier dialects of Lisp, however, used a different kind of scope rule.

Static and Dynamic Scoping

- The scope rules of a language are the rules governing what names (identifiers) mean at each point in a program.
- We call the scope rules of Scheme (and Python)—those that are described by environment diagrams as we’ve been using them—static or lexical scoping.
- But in original Lisp, scoping was dynamic.
- Example (using classic Lisp notation):
  ```lisp
  (defun f (x) ;; Like (define (f x) ...) in Scheme
    (g))
  (defun g ()
    (* x 2))
  (let ((x 3))
    (g)) ;; ==> 6 Using x from (let ((x 3)) ...)
  (f 2) ;; ==> 4 Using x from (defun f (x) ...)
  (g)) ;; ==> 6 Using x from (let ((x 3)) ...)
  ```
- That is, the meaning of \(x\) depends on the most recent and still active definition of \(x\), even where the reference to \(x\) is not nested inside the defining function.
**Remaining Cases**

- We've dealt with function calls, numbers, and symbols.
- This leaves only the **special forms**.
- All special forms lists indicated by their first symbols:
  
  ```lisp
  (quote EXPR) ; Easy: return EXPR unchanged
  (lambda (ARGS) EXPR)
  (define ID EXPR)
  (define (ID ARGS) EXPR)
    ; Same as (define ID (lambda (ARGS) EXPR))
  (if EXPR EXPR-IF-TRUE EXPR-IF-FALSE)
  (begin EXPR1 . . . EXPRn)
  (cond ( (COND-EXPR1 VAL-EXPR1))
       (COND-EXPR2 VAL-EXPR2) . . .)
  (and EXPR, EXPR,...)
  (or EXPR, EXPR,...)
  ```

**Lambda and Functions**

- In the interpreter, evaluating the lambda special form returns a value of some type for representing functions.
- Its content is dictated by what `scheme.apply` will need:
  ```lisp
  (lambda (ARGS) EXPR)
  - The list ARGS.
  - The body EXPR.
  - The parent environment: The environment in which the lambda expression or define that created the function value was evaluated.
  ```

**Other Special Forms**

- Handling the other special forms is pretty straightforward:
- The **if** form is typical: to evaluate

  ```lisp
  (if EXPR-IF-TRUE EXPR-IF-FALSE)
  - Evaluate EXPR.
  - If returned value is false (#t), evaluate EXPR-IF-FALSE and return its value.
  - Otherwise, evaluate EXPR-IF-TRUE and return its value.
  ```

**Getting Iteration via Recursion to Work**

- The interpreter so far uses recursion to get Scheme recursion.
- Doesn't work for long iterations (stack memory overflow).
- As an optional problem, you'll have the chance to complete the **tail-recursion optimization**, where tail calls use (in effect) iteration instead.
What's the Problem?

- Let's look at a very simple tail-recursive loop in Scheme and a call:

  ```scheme
  (define (adder so-far n)
    ; Return SO-FAR + 1 + 2 + 3 + ... + N.
    (if (<= n 0) so-far (adder (+ so-far n) (- n 1))))
  (adder 0 2000)
  ```

- As currently described, our interpreter takes the following steps (indentation shows depth of calls):


  where `[adder]` denotes the function value
  
  ```scheme
  (lambda (so-far n) (if (<= n 0) so-far (adder (+ so-far n) (- n 1))))
  ```

- You can see this rapidly gets out of hand. What to do?

Tail Contexts

- In this function:

  ```scheme
  (define (f x)
    (displayln x)
    (if (> x 0)
        (begin (displayln '+) (* x 2))
        (- x)))
  ```

  - The expressions
    - `(> x 0)`
    - `(displayln '+)`
  
  are not in tail contexts.

  - After they produce their values, some other computation produces the value of the construct that contains them.

Tail Contexts (II)

```scheme
(define (f x)
  (displayln x)
  (if (> x 0)
      (begin (displayln '+) (* x 2))
      (- x)))
```

- The expressions
  - `(> x 0)`
  - `(displayln '+)`

  are not in tail contexts.

- After they produce their values, some other computation produces the value of the construct that contains them.

Crucial Observation

- Consider the functions

  ```scheme
  (define (first x) (some-stuff) (second (+ x 1)) (other-stuff))
  (define (second y) (third y))
  (define (third z) (* z 2))
  ```

  - The call of `third` is in a tail context in `second`.
  
  - Suppose we call `(first 1)`. Normally, `second` would call `third`, which would call `*`.

  - But suppose instead that somehow `second` persuaded `first` to replace its evaluation of `(second 2)` with an evaluation of `(third y)`, but using the local environment set up for the call to `second` (with `y=2`).

  - Since the call to `third` is in a tail context, this replacement must produce the same value as the call to `second`.

  - We call this tail-call optimization: we have effectively removed the call to `second`, so the call only goes two deep, rather than three.

  - In fact, by repeating the process, we can have first replace the calls to `second` and `third` with the evaluation of `(* z 2)` in a local environment with `z=2`. 
Tail-Call Optimization of Tail Recursions

- Let's revisit
  
  \[
  \text{(define (adder so-far n)} \\
  \quad ; \text{Return SO-FAR + 1 + 2 + 3 + \ldots + N.}} \\
  \quad \text{(if (<= n 0) so-far (adder (+ so-far n) (- n 1))))} \\
  \text{(adder 0 2000)} \\
  \]

- Now evaluation can proceed something like this:
  - We call `scheme eval` on `(adder 0 2000)` in the global environment.
  - It tells us to instead call `scheme eval` on 
    \[
    \text{(if (<= n 0) so-far (adder (+ so-far n) (- n 1))))}
    \]
    in an environment with so-far=0, n=2000.
  - That eventually tells us to call `scheme eval` on 
    \[
    \text{(if (<= n 0) so-far (adder (+ so-far n) (- n 1))))}
    \]
  - And so forth.
  - We (i.e., the implementation) don't have to keep track of a whole 
    stack of active recursive function calls.

Tail-Call Optimization in the Project

- As an optional problem, you can make your project do this optimization 
  so that you interpreter will run iterations of arbitrary length.
- Our device for "persuading` scheme eval to replace a call with a 
  different expression is to have it return a special value (of class 
  Unevaluated) that contains an expression that was in a tail context, 
  plus the environment for evaluating that expression.
- If `scheme eval` gets back an Unevaluated object, and needs a real 
  value, it can simply call itself on the expression and environment 
  in that object.