Lecture 26: 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.

- 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)'], ['(define y 42)'])
  >>> b = Buffer(L)
  >>> scheme read(b)
  Pair('define', Pair('x', Pair(Pair('+', Pair('y', Pair(3, nil))), nil))))
  >>> scheme read(b)
  Pair('define', Pair('y', Pair(42, nil)))
  ```

- The interpreter function **scheme apply** has the effect of allowing one to construct and evaluate function calls.

  • Aside: In Python, we've been writing `func(*args)` to get the effect of `apply(func, args)` in ordinary programs.

  • Aside: it is made available to Scheme programmers as the built-in function **apply**:

    ```scheme
    (define L (1 2 3))
    (apply + L) ===> (+ 1 2 3) ===> 6
    ```

- **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 **eval**, and **apply**.

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 · · · 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.

  • • As for **scheme apply**, the evaluation function for Scheme is also available to Scheme programmers, in the form of a function **eval**.

    • E.g., `(eval (list + 1 2))` and `(eval '(+ 1 2))` produce 3.

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 rules. 

- For the calculator, it uses a function **scheme-evaluate** as for the calculator.

- A Scheme interpreter is essentially an extension of the calculator.
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)
  (g))
(defun g ()
  (* x 2))
(setq x 3)
(g) ; ===> 6
(f 2) ; ===> 4
(g) ; ===> 6
```

- 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.

Eval and Scoping

- Dynamic scoping made eval easy to define: interpret any variables according to their "current binding." But eval in pure Scheme behaves like normal functions; it would not have access to the current binding of the place it is called.
- To make it effective, we must introduce the concept of environments. An environment is a data structure that associates identifiers with values. It is called a "frame" or "scope."
- The scope rules of a language are the rules governing what names can be referenced from a given context.

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:
  - 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.

Remaining Cases

- We've dealt with function calls, numbers, and symbols.
- This leaves only the special forms.

Other Special Forms

- Handling the other special forms is pretty straightforward:
  - The if form is typical: to evaluate
    ```lisp
    (if EXPR EXPR-IF-TRUE EXPR-IF-FALSE)
    ```
    
    - Evaluate EXPR.
    - If returned value is false (#f), evaluate EXPR-IF-FALSE and return its value.
    - Otherwise, evaluate EXPR-IF-TRUE and return its value.

Tail-Recursion

- The interpreter so far uses recursion to get Scheme recursion.
- Doesn't work for long iterations (stack memory overflow).
- For extra credit, you'll have the chance to complete the tail-recursion optimization, where tail calls use (in effect) iteration instead.

Other Suggestions

- We've dealt with function calls, numbers, and symbols.
- All special forms must introduce or be introduced by their first symbols.
- This leaves only the special forms.
- We've dealt with function calls, numbers, and symbols.

Lambda and Functions

- The defining function definition of v, x, name the reference to x is not nested inside the lambda.
- That is, the meaning of x depends on the most recent and still active context.

Evaluating Expressions

- However, for the present, we treat names and constants to have the same type:
  - body: an expression (returns a value)
  - body: a lambda (returns a function)
  - body: a variable (returns a value)

Examples (using classic Lisp notation):

- But in other words, the effect of dynamic scoping was dynamic.

- The scope rules of a language are the rules governing what names can be referenced from a given context.