

## 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.

Last modified: Mon Apr 3 13:58:40 2017

CS61A, Lecture #26 1

## 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:

```
>>> from scheme_tokens import tokenize_lines
>>> from buffer import Buffer
>>> L = tokenize_lines(["(define x", " (+ y 3))"])
>>> b = Buffer(L)
>>> b.current()
'('
>>> b.remove_front()
'?'
>>> b.remove_front()
'?'
>>> b.remove_front()
'define'
```

Last modified: Mon Apr 3 13:58:40 2017

CS61A, Lecture #26 2

## scheme\_read

- Finally, the function `scheme_read`, which you will complete, pulls tokens off a `Buffer` until it has a complete Scheme expression:

```
>>> 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)))
```

Last modified: Mon Apr 3 13:58:40 2017

CS61A, Lecture #26 3

## Apply

- The interpreter function `scheme_apply(func, args)` 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`:  

```
(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`.

Last modified: Mon Apr 3 13:58:40 2017

CS61A, Lecture #26 4

## 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 ( $E_0 E_1 \dots E_n$ ) is evaluated by recursively evaluating the  $E_i$  and then using `scheme_apply`.
- But Scheme has a number of other cases to handle.
- **Aside:** As for `scheme_apply`, the evaluation function for Scheme is also available to Scheme programmers, in the form of a function `eval`.
- Eg, `(eval (list + 1 2))` and `(eval '(+ 1 2))` produce 3.

Last modified: Mon Apr 3 13:58:40 2017

CS61A, Lecture #26 5

## 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.

Last modified: Mon Apr 3 13:58:40 2017

CS61A, Lecture #26 6

## 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):

```
(define f (x)
  (g))
(define g ()
  (* x 2))
(setq x 3)
;; Like set! and also defines x at outer level.
(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.

Last modified: Mon Apr 3 13:58:40 2017

CS61A, Lecture #26, 7

## 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 at the place it is called.
- To make it definable (without tricks) in Scheme, one must technically add a parameter to `eval` to convey the desired environment.
- However, for the project, we cheat and arrange to have the environment magically passed into our primitive Scheme `eval` function.

Last modified: Mon Apr 3 13:58:40 2017

CS61A, Lecture #26, 8

## 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:

```
(quote EXPR) ; Easy: return EXPR unchanged
```

```
(lambda (ARGS) EXPR)
(define ID EXPR) EXPR
; Same as (define ID (lambda (ARGS) EXPR))

(if EXPR EXPR-IF-TRUE EXPR-IF-FALSE)
(begin EXPR1 ... EXPRn) ; Evaluate all EXPRi, return last
(cond ( (COND-EXPR1 VAL-EXPR1)
        (COND-EXPR2 VAL-EXPR2) ... )
      (and EXPR1 EXPR2 ... )
      (or EXPR1 EXPR2 ... ))
```

Last modified: Mon Apr 3 13:58:40 2017

CS61A, Lecture #26, 9

## 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:

```
(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.

Last modified: Mon Apr 3 13:58:40 2017

CS61A, Lecture #26, 10

## Other Special Forms

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

```
(if EXPR EXPR-IF-TRUE EXPR-IF-FALSE)
```

  - Evaluate `EXPR`.
  - If returned value is false (`#f`), evaluate `EXPR-IF-FALSE` and re-turn its value.
  - Otherwise, evaluate `EXPR-IF-TRUE` and return its value.

Last modified: Mon Apr 3 13:58:40 2017

CS61A, Lecture #26, 11

## 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.
- Finally, there are many possible suggested extensions for the fun of it (no extra credit is guaranteed: we want you to sleep sometime).

Last modified: Mon Apr 3 13:58:40 2017

CS61A, Lecture #26, 12