Announcements

- HW10 deadline extended to 11:59pm Thursday

- Scheme project out
The user interface to many programming languages is an interactive loop, which

• Reads an expression from the user,
• Parses the input to build an expression tree,
• Evaluates the expression tree,
• Prints the resulting value of the expression

The REPL handles errors by printing informative messages for the user, rather than crashing

A well-designed REPL should not crash on any input!
The Structure of an Evaluator

**Eval**

**Base cases:**
- Primitive values (numbers)
- Look up values bound to symbols

**Recursive calls:**
- Eval(operands) of call expressions
- Apply(operator, arguments)
- Eval(sub-expressions) of special forms

**Apply**

**Base cases:**
- Built-in primitive procedures

**Recursive calls:**
- Eval(body) of user-defined proc's

Requires an environment for name lookup

Creates new environments when applying user-defined procedures
The `scheme_eval` function dispatches on expression form:

- Symbols are bound to values in the current environment
- Self-evaluating primitives are called atoms in Scheme
- All other legal expressions are represented as Scheme lists

```
(define (f s) (if (null? s) '(3) (cons (car s) (f (cdr s)))))
(f (list 1 2))
```
Logical Special Forms

Logical forms may only evaluate some sub-expressions.

• **If** expression: (if <predicate> <consequent> <alternative>)

• **And** and **or**: (and <e₁> ... <eₙ>), (or <e₁> ... <eₙ>)

• **Cond** expr’n: (cond (<p₁> <e₁>) ... (<pₙ> <eₙ>) (else <e>))

The value of an **if** expression is the value of a sub-expression.

- Evaluate the predicate.
- Choose a sub-expression: <consequent> or <alternative>
- Evaluate that sub-expression in place of the whole expression.
Quotation

The **quote** special form evaluates to the quoted expression

\[(quote \ (<expression>))\]

Evaluates to the \(<expression>\) itself, not its value!

'\(<expression>\) is shorthand for \(quote\ <expression>\)

\[(quote (1 2))\]

'\((1 2)\)

The **scheme_read** parser converts shorthand to a combination
Lambda Expressions

Lambda expressions evaluate to user-defined procedures

\[
\text{(lambda \ (<formal-parameters>) \ <body>)}
\]

\[
\text{(lambda \ (x) \ (* \ x \ x))}
\]

class LambdaProcedure(object):

def __init__(self, formals, body, env):
    self.formals = formals  # A scheme list of symbols
    self.body = body        # A scheme expression
    self.env = env          # A Frame instance
A frame represents an environment by having a parent frame.

Frames are Python instances with methods `lookup` and `define`.

In Project 4, Frames do not hold return values.

```
g: Global frame
   y  3
   z  5

[parent=g]
   x  2
   z  4
```
Define Expressions

Define expressions bind a symbol to a value in the first frame of the current environment

\[(\text{define } \text{<name>} \text{ <expression>})\]

Evaluate the \text{<expression>}

Bind \text{<name>} to the result (\text{define} method of the current \text{Frame})

\[(\text{define } x \text{ 2})\]

Procedure definition is a combination of define and lambda

\[(\text{define } (\text{<name>} \text{ <formal parameters>}) \text{ <body>})\]

\[(\text{define } \text{<name>} \text{ (lambda (\text{<formal parameters>}) \text{ <body>})})\]
Applying User-Defined Procedures

Create a new frame in which formal parameters are bound to argument values, whose parent is the env of the procedure.

Evaluate the body of the procedure in the environment that starts with this new frame.

\[
\text{(define (f s) (if (null? s) '(3) (cons (car s) (f (cdr s)))))}
\]

\[
(f \text{ (list 1 2)})
\]
apply[fn;x;a] =

[atom[fn] → [eq[fn;CAR] → caar[x];
               eq[fn;CDR] → cdar[x];
               eq[fn;CONS] → cons[car[x];cadr[x]];]

  eq[fn;ATOM] → atom[car[x]];]

  eq[fn;EQ] → eq[car[x];cadr[x]];]

  T → apply[eval[fn;a];x;a]];]

eq[car[fn];LAMBDA] → eval[caddr[fn];pairlis[cadr[fn];x;a]];]

  eq[car[fn];LABEL] → apply[caddr[fn];x;cons[cons[cadr[fn];
                               caddr[fn]];a]]]

eval[e;a] = [atom[e] → cdr[assoc[e;a]];]

  atom[car[e]] →

    [eq[car[e];QUOTE] → cadr[e];
     eq[car[e];COND] → evcon[cdr[e];a];
     T → apply[car[e];evlis[cdr[e];a];a];]

  T → apply[car[e];evlis[cdr[e];a];a]]
Dynamic Scope

The way in which names are looked up in Scheme and Python is called *lexical scope* (or *static scope*)

**Lexical scope**: The parent of a frame is the environment in which a procedure was *defined*

**Dynamic scope**: The parent of a frame is the environment in which a procedure was *called*

\[
\text{(define f (lambda (x) (+ x y)))}
\]

\[
\text{(define g (lambda (x y) (f (+ x x))))}
\]

\[
\text{(g 3 7)}
\]

**Lexical scope**: The parent for f's frame is the global frame

*Error: unknown identifier: y*

**Dynamic scope**: The parent for f's frame is g's frame

13