Announcements
Interpreting Scheme
The Structure of an Interpreter

**Eval**

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

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

**Apply**

Base cases:
- Built-in primitive procedures

Recursive calls:
- Eval(body) of user-defined procedures

Requires an environment for symbol lookup

Creates a new environment each time a user-defined procedure is applied
Special Forms
Scheme Evaluation

The `scheme_eval` function chooses behavior based on expression form:

- Symbols are looked up in the current environment
- Self-evaluating expressions are returned as values
- All other legal expressions are represented as Scheme lists, called combinations

\[
\begin{align*}
\text{(if } & <\text{predicate}> <\text{consequent}> <\text{alternative}>) \\
\text{(lambda} & (<\text{formal-parameters}>) <\text{body}>) \\
\text{(define} & <\text{name}> <\text{expression}>) \\
(<\text{operator}>& <\text{operand }0> \ldots <\text{operand }k>)
\end{align*}
\]

Any combination that is not a known special form is a call expression

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

\[
(\text{demo } (\text{list } 1 2))
\]
Logical Forms
Logical Special Forms

Logical forms may only evaluate some sub-expressions

- **If** expression: \((\text{if } \langle\text{predicate}\rangle \ \langle\text{consequent}\rangle \ \langle\text{alternative}\rangle)\)
- **And** and **or**: \((\text{and } \langle e_1 \rangle \ldots \langle e_n \rangle), \ (\text{or } \langle e_1 \rangle \ldots \langle e_n \rangle)\)
- **Cond** expression: \((\text{cond } (\langle p_1 \rangle \ \langle e_1 \rangle) \ldots (\langle p_n \rangle \ \langle e_n \rangle) \ (\text{else } \langle e \rangle))\)

The value of an if expression is the value of a sub-expression:

- Evaluate the predicate
- Choose a sub-expression: \langle\text{consequent}\rangle or \langle\text{alternative}\rangle
- Evaluate that sub-expression to get the value of the whole expression

(Demo)
Quotation
**Quotation**

The quote special form evaluates to the quoted expression, which is not evaluated

```
(quote <expression>)  (quote (+ 1 2))  evaluates to the three-element Scheme list (+ 1 2)
```

The `<expression>` itself is the value of the whole quote expression

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

```
(quote (1 2))  is equivalent to  '(1 2)
```

The `scheme_read` parser converts shorthand ' to a combination that starts with quote

```
(Demo)
```
Lambda Expressions
Lambda Expressions

Lambda expressions evaluate to user-defined procedures

```
(lambda (<formal-parameters>) <body>)

(lambda (x) (* x x))
```

class LambdaProcedure:
    def __init__(self, formals, body, env):
        self.formals = formals ........................................... A scheme list of symbols
        self.body = body .......................................................... A scheme list of expressions
        self.env = env .................................................................. A Frame instance
Frames and Environments

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

f1: [parent=g]
    x 2
    z 4
```

(Demo)
Define Expressions
Define Expressions

Define binds a symbol to a value in the first frame of the current environment.

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

1. Evaluate the \text{expression}

2. Bind \text{name} to its value in the current frame

\[
\text{(define } x \text{ (+ 1 2))}
\]

Procedure definition is shorthand of define with a lambda expression

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

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

To apply a user-defined procedure, create a new frame in which formal parameters are bound to argument values, whose parent is the env attribute of the procedure.

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

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

\[
\text{(demo (list 1 2))}
\]
Eval/Apply in Lisp 1.5

\[
\text{apply}[\text{fn};x;a] = \\
[\text{atom}[\text{fn}] \rightarrow \text{eq}[\text{fn};\text{CAR}] \rightarrow \text{caar}[x]; \\
\text{eq}[\text{fn};\text{CDR}] \rightarrow \text{cdar}[x]; \\
\text{eq}[\text{fn};\text{CONS}] \rightarrow \text{cons}[	ext{car}[x];\text{cadr}[x]]; \\
\text{eq}[\text{fn};\text{ATOM}] \rightarrow \text{atom}[	ext{car}[x]]; \\
\text{eq}[\text{fn};\text{EQ}] \rightarrow \text{eq}[	ext{car}[x];\text{cadr}[x]]; \\
\text{T} \rightarrow \text{apply}[\text{eval}[\text{fn};a];x;a]; \\
\text{eq}[	ext{car}[\text{fn}];\text{LAMBDA}] \rightarrow \text{eval}[	ext{caddr}[\text{fn}];\text{pairlis}[	ext{cadr}[\text{fn}];x;a]]; \\
\text{eq}[	ext{car}[\text{fn}];\text{LABEL}] \rightarrow \text{apply}[	ext{caddr}[\text{fn}];x;\text{cons}[	ext{cons}[	ext{cadr}[\text{fn}]; \\
\text{caddr}[\text{fn}]];a)]; \\
\]

\[
\text{eval}[e;a] = [\text{atom}[e] \rightarrow \text{cdr}[	ext{assoc}[e;a]]; \\
\text{atom}[	ext{car}[e]] \rightarrow \\
[\text{eq}[	ext{car}[e];\text{QUOTE}] \rightarrow \text{cdr}[e]; \\
\text{eq}[	ext{car}[e];\text{COND}] \rightarrow \text{evcon}[	ext{cdr}[e];a]; \\
\text{T} \rightarrow \text{apply}[	ext{car}[e];\text{evlis}[	ext{cdr}[e];a];a]; \\
\text{T} \rightarrow \text{apply}[	ext{car}[e];\text{evlis}[	ext{cdr}[e];a];a]) \\
\]