61A Lecture 26

Wednesday, November 6
Announcements

• Project 1 composition revisions due Thursday 11/7 @ 11:59pm.
• Homework 8 due Tuesday 11/12 @ 11:59pm, and it's in Scheme!
• Project 4 due Thursday 11/21 @ 11:59pm, and it's a Scheme interpreter!

New Policy: An improved final exam score can make up for low midterm scores.

• If you scored less than 60/100 midterm points total, then you can earn some points back.
• You don't need a perfect score on the final to do so.
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
The scheme_eval function dispatches on expression form:

• Symbols are bound to values in the current environment.
• Self-evaluating expressions are returned.
• All other legal expressions are represented as Scheme lists, called *combinations*.

```
(if <predicate> <consequent> <alternative>)
(lambda (<formal-parameters>) <body>)
(define <name> <expression>)
(<operator> <operand 0> ... <operand k>)
```

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

```scheme
(define (demo s) (if (null? s) '(3) (cons (car s) (demo (cdr s)))))
```

```
(demo (list 1 2))
```
Logical Forms
Logical Special Forms

Logical forms may only evaluate some sub-expressions.

- **If** expression:  
  \[(\text{if } \text{<predicate>} \text{ <consequent>} \text{ <alternative>})\]

- **And** and **or**:  
  \[(\text{and } \text{<e}_1\ldots \text{<e}_n\text{>}), \quad (\text{or } \text{<e}_1\ldots \text{<e}_n\text{>})\]

- **Cond** expr’n:  
  \[(\text{cond (}<p_1\text{<e}_1\text{>}) \ldots (\text{<p}_n\text{<e}_n\text{>) (else } \text{<e>}))\]

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
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.

(Demo)
Lambda Expressions
**Lambda Expressions**

Lambda expressions evaluate to user-defined procedures.

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

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

class LambdaProcedure:

def __init__(self, formals, body, env):
    self.formals = formals
    self.body = body
    self.env = env

A scheme list of symbols
A scheme expression
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.

```
<table>
<thead>
<tr>
<th>g: Global frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
</tr>
<tr>
<td>z</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>f1: [parent=g]</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
</tr>
<tr>
<td>z</td>
</tr>
</tbody>
</table>
```

(Demo)
Define Expressions
Define Expressions

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

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

1. Evaluate the <expression>.

2. Bind <name> to its value in the current frame.

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

Procedure definition is shorthand of define with a lambda expression.

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

\[
\text{(define } \text{<name> } \text{(lambda } \text{<formal parameters> } \text{<body>})}
\]
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 of the procedure.

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

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

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

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

\[
\text{eval}[e;a] = [\text{atom}[e] \rightarrow \text{cdr}[\text{assoc}[e;a]]; \\
\quad \text{atom}[\text{car}[e]] \rightarrow \\
\quad \quad [\text{eq}[\text{car}[e];\text{QUOTE}] \rightarrow \text{cadr}[e]; \\
\quad \quad \text{eq}[\text{car}[e];\text{COND}] \rightarrow \text{evcon}[\text{cdr}[e];a]; \\
\quad \quad T \rightarrow \text{apply}[\text{car}[e];\text{evlis}[\text{cdr}[e];a];a]]; \\
\quad T \rightarrow \text{apply}[\text{car}[e];\text{evlis}[\text{cdr}[e];a];a]]
\]
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*.

```scheme
(define f (lambda (x) (+ x y)))
(define g (lambda (x y) (f (+ x x))))
(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

Special form to create dynamically scoped procedures