61A Lecture 27

Wednesday, November 5

Announcements

- Homework 7 due Wednesday 11/5 @ 11:59pm
- Project 1 composition revisions due Wednesday 11/5 @ 11:59pm
- Copy hop.py to an instructional server and run: submit proj1revision
- Quiz 2 released Wednesday 11/5 & due Thursday 11/6 @ 11:59pm
  - Open note, open interpreter, closed classmates, closed Internet
- Midterm survey due Monday 11/10 @ 11:59pm (Thanks!)
- Project 4 due Thursday 11/20 @ 11:59pm (Big!)

The Structure of an Interpreter

Interpreting Scheme

Scheme Evaluation

Special Forms

Logical Forms

Logical Special Forms

Requires an environment for symbol lookup

Creates a new environment each time a user-defined procedure is applied

Base cases:
  - Primitive values (numbers)
  - Look up values bound to symbols
  - Built-in primitive procedures

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

Recursive calls:
  - Eval(sub-expressions) of special forms

Requires an environment for symbol lookup

Create a new environment each time a user-defined procedure is applied

Evaluate

Apply

Base cases:
  - Built-in primitive procedures
  - Eval(body) of user-defined procedures

Special forms are identified by the first list element

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

Special forms

(if <predicate> <consequent> <alternative>)

(lambda <formal-parameters> <body>)

(define <name> <expression>)

(operator <operator> <operands>)

The scheme_eval function dispatches 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

Logical forms may only evaluate some sub-expressions.

- If expression: (if <predicate> <consequent> <alternative>)
- And and or: (and <e1> ... <en>), (or <e1> ... <en>)
- Cond expression: (cond (<p1> <e1>) ... (<pn> <en>) (else <e>))

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

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

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

```
(quote <expression>)  \equiv  \text{<expression>}
```

The `<expression>` itself is the value of the whole quote expression. `'<expression>' is shorthand for `(quote <expression>)`

```
'(1 2)  \equiv  \text{(quote (1 2))}
```

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

```
(Demo)
```

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  \equiv  \text{A scheme list of symbols}
    self.body = body  \equiv  \text{A scheme expression}
    self.env = env  \equiv  \text{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>Global frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>y 3</td>
</tr>
<tr>
<td>z 5</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>(parent: g)</td>
</tr>
<tr>
<td>x 2</td>
</tr>
<tr>
<td>z 4</td>
</tr>
</tbody>
</table>

frames

(Demo)

Define Expressions

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

```
(define <name> <expression>)
```

1. Evaluate the `<expression>`
2. Bind `<name>` to its value in the current frame

```
(define x (+ 1 2))
```

Procedure definition is shorthand of define with a lambda expression

```
(define <name> (lambda <formal parameters> <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.

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

```
(demo (list 1 2))
```

```
<table>
<thead>
<tr>
<th>Global frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>demo</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>(parent: g)</td>
</tr>
<tr>
<td>['s' 1]</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>[parent: g]</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>[parent: g]</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
Eval/Apply in Lisp 1.5

```lisp
apply[fn;x] =
    [atom[fn] - (eq[fn,CAR] = car[x])]
    eq(fn,CONSP] = cons[car[x];cdr[x]]
    eq[fn;ATOM] = atom[car[x]]
    eq[fn,PS] = eq[car[x];cdr[x]]
    T - apply[eval[fn];a;x]]

    eq[car[fn];LAMBDA] - eval[cadr[fn]];pairfirst[cadr[fn];x;a]]
    eq[car[fn];LABEL] - apply[cons[car[fn];
                               cadr[fn]];a]]

eval[x] = [atom[x] = cadr[assoc[x;a]]]
    atom[car[x]] =
        eq[car[x];QUOTE] = cadr[x]
    eq[car[x];COND] = evcon[cdr[x];a]
    T - apply[car[x];evcon[cdr[x];a]]
```

T - apply[car[x];evcon[cdr[x];a]]