61A Lecture 26

Announcements

Scheme Recursive Art Contest: Start Early!

Interpreting Scheme

The Structure of an Interpreter

Special Forms

Scheme Evaluation

Logical Forms
**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** expression: `(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 to get the value of the whole expression

**Quotation**

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

`(quote <expression>)`

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

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

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

**Lambda Expressions**

Lambda expressions evaluate to user-defined procedures:

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

```python
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:

<table>
<thead>
<tr>
<th>Global Frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>x: 1</td>
</tr>
<tr>
<td>y: 5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parent Frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>s: 2</td>
</tr>
<tr>
<td>z: 4</td>
</tr>
</tbody>
</table>

**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>))`
- `(define x (lambda (x) (* x x)))`
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) "" (cons (car s) (demo (cdr s)))))
```

```
(define (demo list) 1)
1 Pair 2 Pair nil
```

Eval/Apply in Lisp 1.5

```
apply[fn.x.s] = [atom[fn] = atom[x] = car[s];
    eq[fn,CAR] = car[x];
    eq[fn,CDDR] = cons(car[x];cdr[s]);
    eq[fn,ATOM] = atom(car[x]);
    eq[fn,.remaining] = eval[expression[x]]];
T = apply[apply[eval[expression[x]]]];
```

```
eval[expression[x]] = [atom[x] = cdr[assoc[x]];`
    atom[car[x]] = [eq[car[x];QUOTE] = car[x];
    eq[car[x];CDDR] = evcon[cdr[x]];]
T = apply[apply[apply[expression[x]]]]
```