Scheme Recursive Art Contest: Start Early!

Fall 2012 Featherweight Winner
176 Scheme Tokens

Fall 2013 Heavyweight Winner
1857 Scheme Tokens

Extra lecture on ray tracing
Monday 11/2 6:30pm
A1 Hearst Annex
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

\[
\text{if} \ <\text{predicate}> \ <\text{consequent}> \ <\text{alternative}>
\]

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

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

\[
<\text{operator}> \ <\text{operand 0}> \ldots \ <\text{operand k}>
\]

Special forms are identified by the first list element.

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: \((\text{if } \text{<predicate>} \text{<consequent>} \text{<alternative>})\)
- **And** and **or**: \((\text{and } \text{<e1>} ... \text{<en>}), \quad (\text{or } \text{<e1>} ... \text{<en>})\)
- **Cond** expression: \((\text{cond } (\text{<p1> } \text{<e1>}) ... (\text{<pn> } \text{<en>}) (\text{else } \text{<e>}))\)

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 to get the value of the whole expression

(Demo)
Quotation
Quotation

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

\[(\text{quote } \text{<expression>})\] \quad \text{(quote } (+ 1 2)) \quad \text{evaluates to the three-element Scheme list} \quad (+ 1 2)

The \text{<expression>} itself is the value of the whole quote expression

\'\text{<expression>} is shorthand for \text{(quote } \text{<expression>})

\[(\text{quote } (1 2))\] \quad \text{is equivalent to} \quad '(1 2)

The \text{scheme_read parser converts shorthand } \text{'} \text{ to a combination that starts with } \text{quote}

\[(\text{Demo})\]
Lambda Expressions
Lambda Expressions

Lambda expressions evaluate to user-defined procedures

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

\[
\text{lambda } \left( x \right) \left( * \; x \; x \right)
\]

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>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 \(<\text{expression}>\)

2. Bind \(<\text{name}>\) to its value in the current frame

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

Procedure definition is shorthand of \text{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` 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))
```
Eval/Apply in Lisp 1.5

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]]}