Interpreting Scheme

The Structure of an Interpreter

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

Requires an environment for symbol lookup
Creates a new environment each time a user-defined procedure is applied

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

Special forms are identified by the first list element
Any combination that is not a known special form is a call expression

Logical Forms

Logical forms may only evaluate some sub-expressions
• If expression: \[
    \text{if} (\text{predicate} \ \text{consequent} \ \text{alternative})
\]
• And and or: \[
    \text{and} \ \text{e1} ... \ \text{en}, \ \text{or} \ \text{e1} ... \ \text{en}
\]
• Cond expression: \[
    \text{cond} (\text{e1} \ \text{e1}) ... (\text{en} \ \text{en}) (\text{else} \ e)
\]

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 in place of the whole expression.
Quotation

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

\[ \text{quote} \langle \text{expression} \rangle \]

The \langle expression \rangle itself is the value of the whole quote expression.

\'\langle expression \rangle is shorthand for (quote \langle expression \rangle)
\[ \text{quote} (\langle 1 2 \rangle) \]

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

(Demo)

Lambda Expressions

Lambda expressions evaluate to user-defined procedures:

\[ \lambda (\langle \text{formal parameters} \rangle) \langle \text{body} \rangle \]

\[ \lambda (x) (* x x) \]

Class LambdaProcedure:

\[ \text{class} \Lambda\text{Procedure}: \]

\[ \text{def} \_\_init\_\_(self, \text{formals}, \text{body}, \text{env}): \]

\[ \text{self.formals} = \text{formals}; \text{A scheme list of symbols} \]

\[ \text{self.body} = \text{body}; \text{A scheme expression} \]

\[ \text{self.env} = \text{env}; \text{A Frame instance} \]

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.

Define Expressions

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

1. Evaluate the \langle expression \rangle
2. Bind \langle name \rangle to its value in the current frame

\[ \text{define} \langle \text{name} \rangle \langle \text{expression} \rangle \]

Procedure definition is shorthand of define with a lambda expression

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

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.
Eval/Apply in Lisp 1.5

apply[n,x,a] =
  [atom[n] - (eq[n, CAR] = car[x])
   eq[n, CDR] = cadr[x]
   eq[n, CONS] = cons[car[x]; cadr[x]]
   eq[n, ATOM] = atom[car[x]]
   eq[n, EQ] = eq[car[x]; cadr[x]]
   T = apply[eval[n], a, x]]

  eq[car[n], LAMBDA] - eval[cadr[n]; (package[cadr[n]; x, a])]
  eq[car[n], LABEL] - apply[cadr[n]; (package[cadr[n];
                                   cadr[n][a]])]

  eval[n, a] = [atom[x] = cdr[assoc[x], a]]
  atom[car[x]] -
    [eq[car[x], QUOTE] = cadr[x]]
  eq[car[x], COND] - evcon[cdrt[x]; a]
  T = apply[car[x]; evlist[cdrt[x]; a]]
  T = apply[car[x]; evlist[cdrt[x]; a]]