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

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 choose 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

```scheme
(if <predicate> <consequent> <alternative>)
(and <e1> ... <en>),    (or <e1> ... <en>)
(cond (<p1> <e1>) ... (<pn> <en>) (else <e>))
```

Special forms are identified by the first list element

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 Special Forms

Logical special 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

```scheme
(demo)
```
Quotation

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

\[(quote \text{ <expression>})\]

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

"<expression>" is shorthand for \[(quote \text{ <expression>})\]

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

(Demo)

Lambda Expressions

Lambda expressions evaluate to user-defined procedures

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

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

class LambdaProcedure:

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

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

\[\text{self.body} = \text{body} \ \text{-------------------A scheme list of expressions}\]

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

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.

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

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 define with a lambda expression

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

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

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

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

(Demo list 1 2)
Eval/Apply in Lisp 1.5

apply[nx; a] =
[a] = [atom[nx] = eq[nx,CAR] = car[nx] ];
eq[nx,CDR] = car[nx];
eq[nx,CONS] = cons[car[nx];cdr[nx] ];
eq[nx,ATOM] = atom[car[nx]]; eq[nx,EQ] = eq[car[nx];cdr[nx] ];
T = apply[eval[nx];a];
eq[car[nx],LAMBDA] = eval[cdr[car[nx]];pextract[cdr[nx];x];]);
eq[car[nx],LABEL] = apply[cadr[car[nx];x];cons[cons[cons[cons[car[nx];cdr[nx] ];
   car[nx];a]];]
eval[x] = [atom[x] = or[assoc[x];x];]
   atom[car[x]] =
eq[car[x],QUOTE] = car[x];
eq[car[x],COND] = evcon[or[or[or[car[x];a];
   T = apply[or[or[or[car[x];evcon[or[or[or[car[x];a];
   a];a];a];a];a];a] ]]}