Interpreters
Announcements
Interpreting Scheme
The Structure of an Interpreter
The Structure of an Interpreter

Eval

Apply
The Structure of an Interpreter

Base cases: $Eval$

Apply
The Structure of an Interpreter

Base cases:
• Primitive values (numbers)
The Structure of an Interpreter

Base cases:
- Primitive values (numbers)

Recursive calls:
Base cases:
• Primitive values (numbers)

Recursive calls:
• Eval(operator, operands) of call expressions
The Structure of an Interpreter

Base cases:
• Primitive values (numbers)

Recursive calls:
• Eval(operator, operands) of call expressions
• Apply(procedure, arguments)
The Structure of an Interpreter

Base cases:
- Primitive values (numbers)

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

Base cases:
- Built-in primitive procedures
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)

Base cases:
- Built-in primitive procedures
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)

**Apply**

Base cases:
- Built-in primitive procedures
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
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
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
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

---

Requires an environment for symbol lookup

**Apply**

Base cases:
- Built-in primitive procedures

Recursive calls:
- Eval(body) of user-defined procedures
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

Base cases:
• Built-in primitive procedures

Recursive calls:
• Eval(body) of user-defined procedures

Creates a new environment each time a user-defined procedure is applied
Special Forms
Scheme Evaluation
Scheme Evaluation

The scheme_eval function choose behavior based on expression form:
Scheme Evaluation

The scheme_eval function choose behavior based on expression form:
• Symbols are looked up in the current environment
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
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 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

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

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

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

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

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

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

\[(\text{if} \ <\text{predicate}> \ <\text{consequent}> \ <\text{alternative}>)\]
\[(\text{lambda} \ (<\text{formal-parameters}>) \ <\text{body}>)\]
\[(\text{define} \ <\text{name}> \ <\text{expression}>)\]
\[(<\text{operator}> \ <\text{operand} 0> \ ... \ <\text{operand} k>)\]
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

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

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

(define <name> <expression>)

(<operator> <operand 0> ... <operand k>)
```

Special forms are identified by the first list element.
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

\[
\begin{align*}
&\text{(if} \ <\text{predicate}> \ <\text{consequent}> \ <\text{alternative}>\text{)} \\
&(\text{lambda} \ (<\text{formal-parameters}>)) \ <\text{body}> \\
&(\text{define} \ <\text{name}> \ <\text{expression}>)
\end{align*}
\]

Special forms are identified by the first list element.
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> \ ... \ <\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.
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

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

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

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

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

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

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 Special Forms

Logical forms may only evaluate some sub-expressions
Logical Special Forms

Logical forms may only evaluate some sub-expressions

• **If** expression: \((\text{if} \ <\text{predicate}> \ <\text{consequent}> \ <\text{alternative}>)\)
Logical Special Forms

Logical forms may only evaluate some sub-expressions

- **If** expression: \((\text{if } \text{<predicate>} \ \text{<consequent>} \ \text{<alternative>})\)
- **And** and **or**: \((\text{and } \text{<e1>} \ ... \ \text{<en>}), \ (\text{or } \text{<e1>} \ ... \ \text{<en>})\)
Logical forms may only evaluate some sub-expressions

- **If** expression: \( (\text{if} \ <\text{predicate}> \ <\text{consequent}> \ <\text{alternative}>) \)
- **And** and **or**: \( (\text{and} \ <\text{e1}> \ldots \ <\text{en}>) \), \( (\text{or} \ <\text{e1}> \ldots \ <\text{en}>) \)
- **Cond** expression: \( (\text{cond} (\langle p1 \rangle \ <\text{e1}>) \ldots (\langle pn \rangle \ <\text{en}>)(\text{else} \ <\text{e}>)\) \)
Logical Special Forms

Logical forms may only evaluate some sub-expressions

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

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

Logical forms may only evaluate some sub-expressions

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

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

- Evaluate the predicate
Logical Special Forms

Logical forms may only evaluate some sub-expressions

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

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

- Evaluate the predicate
- Choose a sub-expression: \(<\text{consequent}> \ or \ <\text{alternative}>\)
Logical Special Forms

Logical forms may only evaluate some sub-expressions

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

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
Logical Special Forms

Logical forms may only evaluate some sub-expressions

- **If** expression: \( (\text{if} \ <\text{predicate}> \ <\text{consequent}> \ <\text{alternative}>) \)
- **And** and **or**: \( (\text{and} \ <e_1> ... <e_n>), \ (\text{or} \ <e_1> ... <e_n>) \)
- **Cond** expression: \( (\text{cond} \ (<p_1> <e_1>) ... (<p_n> <e_n>) \ (\text{else} <e>)) \)

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 to get the value of the whole expression

(Demo)
Quotation
The quote special form evaluates to the quoted expression, which is not evaluated
Quotation

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

(quote <expression>)
Quotation

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

\[(quote \langle\text{expression}\rangle) \quad (quote \ (+\ 1\ 2))\]

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

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

\[
(quote \ <\text{expression}>\ ) \quad (quote \ (+ \ 1 \ 2))
\]

The \texttt{quote} itself is the value of the whole quote expression

\[
\text{evaluates to the three-element Scheme list} \quad (+ \ 1 \ 2)
\]
The quote special form evaluates to the quoted expression, which is not evaluated

\[(\text{quote } \langle \text{expression} \rangle) \quad (\text{quote } (+ 1 2))\]

evaluates to the three-element Scheme list

\[(+ 1 2)\]

The \langle \text{expression} \rangle itself is the value of the whole quote expression

\[\langle \text{expression} \rangle\] is shorthand for (quote \langle \text{expression} \rangle)
The quote special form evaluates to the quoted expression, which is not evaluated

\[
(quote \ <expression>) \quad (quote \ (+ \ 1 \ 2))
\]

(quote (+ 1 2)) evaluates to the three-element Scheme list

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

'\<expression\> is shorthand for (quote <expression>)

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

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

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

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

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

\[(quote\,(1\,2))\] is equivalent to \'(1\,2)

The scheme_read parser converts shorthand ' to a combination that starts with quote
The quote special form evaluates to the quoted expression, which is not evaluated

```scheme
(quote <expression>)  (quote (+ 1 2)) evaluates to the three-element Scheme list  (+ 1 2)
```

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

'`<expression>` is shorthand for (quote `<expression>`)

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

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

```scheme
(Demo)
```
Lambda Expressions
Lambda Expressions
Lambda Expressions

Lambda expressions evaluate to user-defined procedures
Lambda Expressions

Lambda expressions evaluate to user-defined procedures

(lambda (<formal-parameters>) <body>)
Lambda Expressions

Lambda expressions evaluate to user-defined procedures

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

(l lambda (x) (* x x))
Lambda Expressions

Lambda expressions evaluate to user-defined procedures

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

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

```python
class LambdaProcedure:
    def __init__(self, formals, body, env):
        self.formals = formals
        self.body = body
        self.env = env
```
Lambda Expressions

Lambda expressions evaluate to user-defined procedures

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

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

class LambdaProcedure:
    def __init__(self, formals, body, env):
        self.formals = formals  # A scheme list of symbols
        self.body = body
        self.env = env
Lambda Expressions

Lambda expressions evaluate to user-defined procedures

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

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

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
Lambda Expressions

Lambda expressions evaluate to user-defined procedures

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

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

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
Frames and Environments

A frame represents an environment by having a parent frame.
Frames and Environments

A frame represents an environment by having a parent frame

Frames are Python instances with methods `lookup` and `define`
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.
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.

```
g: Global frame
    y  3
    z  5
```
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.

```
g: Global frame
  y | 3
  z | 5

f1: [parent=g]
  x | 2
  z | 4
```
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.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>g:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>z</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>f1:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>z</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>
Define Expressions
Define Expressions
Define Expressions

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

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

(define <name> <expression>)
Define Expressions

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

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

1. Evaluate the <expression>
Define Expressions

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

(\texttt{define } \texttt{name} \texttt{ expression})

1. Evaluate the \texttt{expression}

2. Bind \texttt{name} to its value in the current frame
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 <expression>
2. Bind <name> to its value in the current frame

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

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

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

Procedure definition is shorthand of 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
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.
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.
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)))))
```
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)))))
```

```
g: Global frame
  demo |_
      LambdaProcedure instance [parent=g]
```
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))
```

```
g: Global frame
demo | LambdaProcedure instance [parent=g]
```
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.

\[
\begin{align*}
\text{(define } & \text{(demo } s) \text{ (if (null? } s) \text{ '}(3) \text{ (cons (car } s) \text{ (demo (cdr } s))))) \\
& \text{(demo (list 1 2))}
\end{align*}
\]
## 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))
```
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))
```
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
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]]