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)

Eval

Apply
The Structure of an Interpreter

Base cases:
• Primitive values (numbers)

Recursive calls:
The Structure of an Interpreter

Base cases:
• Primitive values (numbers)

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

Base cases:  
• Primitive values (numbers)

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

**Eval**

Base cases:
- Primitive values (numbers)

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

**Apply**

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(operands) of call expressions
• Apply(operator, 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(operands) of call expressions  
- Apply(operator, arguments)  
- Eval(sub-expressions) of special forms

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(operands) of call expressions
- Apply(operator, arguments)
- Eval(sub-expressions) of special forms

Eval

Base cases:
- Built-in primitive procedures

Recursive calls:
- Eval(body) of user-defined proc's

Apply
The Structure of an Interpreter

Base cases:
- Primitive values (numbers)
- Look up values bound to symbols

Recursive calls:
- Eval(operands) of call expressions
- Apply(operator, arguments)
- Eval(sub-expressions) of special forms

Eval

Base cases:
- Built-in primitive procedures

Recursive calls:
- Eval(body) of user-defined proc's

Apply
The Structure of an Interpreter

**Eval**

Base cases:
- Primitive values (numbers)
- Look up values bound to symbols

Recursive calls:
- Eval(operands) of call expressions
- Apply(operator, arguments)
- Eval(sub-expressions) of special forms

**Apply**

Base cases:
- Built-in primitive procedures

Recursive calls:
- Eval(body) of user-defined proc's

Requires an environment for name lookup
The Structure of an Interpreter

**Eval**

Base cases:
- Primitive values (numbers)
- Look up values bound to symbols

Recursive calls:
- Eval(operands) of call expressions
- Apply(operator, arguments)
- Eval(sub-expressions) of special forms

**Apply**

Base cases:
- Built-in primitive procedures

Recursive calls:
- Eval(body) of user-defined proc's

Requires an environment for name lookup

Creates new environments when applying user-defined procedures
Scheme Evaluation
Scheme Evaluation

The scheme_eval function dispatches on expression form:
Scheme Evaluation

The scheme_eval function dispatches on expression form:

- Symbols are bound to values in the current environment.
Scheme Evaluation

The scheme_eval function dispatches on expression form:
• Symbols are bound to values in the current environment.
• Self-evaluating primitives are called atoms in Scheme.
Scheme Evaluation

The scheme_eval function dispatches on expression form:
• Symbols are bound to values in the current environment.
• Self-evaluating primitives are called *atoms* in Scheme.
• All other legal expressions are represented as Scheme lists.
Scheme Evaluation

The `scheme_eval` function dispatches on expression form:

- Symbols are bound to values in the current environment.
- Self-evaluating primitives are called *atoms* in Scheme.
- All other legal expressions are represented as Scheme lists.

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

The scheme_eval function dispatches on expression form:
• Symbols are bound to values in the current environment.
• Self-evaluating primitives are called atoms in Scheme.
• All other legal expressions are represented as Scheme lists.

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

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

The scheme_eval function dispatches on expression form:

- Symbols are bound to values in the current environment.
- Self-evaluating primitives are called atoms in Scheme.
- All other legal expressions are represented as Scheme lists.

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

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

(define <name> <expression>)
```
Scheme Evaluation

The `scheme_eval` function dispatches on expression form:

• Symbols are bound to values in the current environment.
• Self-evaluating primitives are called *atoms* in Scheme.
• All other legal expressions are represented as Scheme lists.

\[
  (\text{if } \langle\text{predicate}\rangle \ \langle\text{consequent}\rangle \ \langle\text{alternative}\rangle)
\]

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

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

\[
  (\langle\text{operator}\rangle \ \langle\text{operand 0}\rangle \ \ldots \ \langle\text{operand k}\rangle)
\]
Scheme Evaluation

The scheme_eval function dispatches on expression form:
• Symbols are bound to values in the current environment.
• Self-evaluating primitives are called atoms in Scheme.
• All other legal expressions are represented as Scheme lists.

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

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

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

\[
\text{(operator} \ <\text{operand 0}> \ ... \ <\text{operand k}>\text{)}
\]
The scheme_eval function dispatches on expression form:

- Symbols are bound to values in the current environment.
- Self-evaluating primitives are called *atoms* in Scheme.
- All other legal expressions are represented as Scheme lists.

\[
\text{(if} \ <\text{predicate}> \ <\text{consequent}> \ <\text{alternative}>\)
\]
\[
\text{(lambda} \ (<\text{formal-parameters}>) \ <\text{body}>\)
\]
\[
\text{(define} \ <\text{name}> \ <\text{expression}>\)
\]
\[
(<\text{operator}> \ <\text{operand 0}> \ ... \ <\text{operand k}>)
\]
The scheme_eval function dispatches on expression form:
- Symbols are bound to values in the current environment.
- Self-evaluating primitives are called *atoms* in Scheme.
- All other legal expressions are represented as Scheme lists.

- \[(\text{if} \ <\text{predicate}> \ <\text{consequent}> \ <\text{alternative}>)\]
- \[(\text{lambda} \ (<\text{formal-parameters}>) \ <\text{body}>)\]
- \[(\text{define} \ <\text{name}> \ <\text{expression}>)\]
- \[(<\text{operator}> \ <\text{operand 0}> \ ... \ <\text{operand k}>)]\]
The scheme_eval function dispatches on expression form:
- Symbols are bound to values in the current environment.
- Self-evaluating primitives are called *atoms* in Scheme.
- All other legal expressions are represented as Scheme lists.

```
(define (f s) (if (null? s) '(3) (cons (car s) (f (cdr s)))))
```
Scheme Evaluation

The scheme_eval function dispatches on expression form:

- Symbols are bound to values in the current environment.
- Self-evaluating primitives are called *atoms* in Scheme.
- All other legal expressions are represented as Scheme lists.

```
(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

Anything not a known special form is a call expression

```
(define (f s) (if (null? s) '(3) (cons (car s) (f (cdr s)))))
(f (list 1 2))
```
The `scheme_eval` function dispatches on expression form:
- Symbols are bound to values in the current environment.
- Self-evaluating primitives are called *atoms* in Scheme.
- All other legal expressions are represented as Scheme lists.

```
(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.

Anything not a known special form is a call expression.

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

```
(f (list 1 2))
```

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

Logical forms may only evaluate some sub-expressions.

• **If** expression: `(if <predicate> <consequent> <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**: \((\text{and} \ <e_1> \ldots \ <e_n>), \quad (\text{or} \ <e_1> \ldots \ <e_n>)\)
Logical Special Forms

Logical forms may only evaluate some sub-expressions.

- **If** expression:  
  \[(\text{if} \ <\text{predicate}> \ <\text{consequent}> \ <\text{alternative}>)\]

- **And** and **or**:  
  \[(\text{and} \ <e_1> \ ... \ <e_n>), \quad (\text{or} \ <e_1> \ ... \ <e_n>)\]

- **Cond expr’n**:  
  \[(\text{cond} \ (<p_1> \ <e_1>) \ ... \ (<p_n> \ <e_n>) \ (\text{else} \ <e>))\]
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 expr'n**:
  - (cond (<p₁> <e₁>) ... (<pₙ> <eₙ>) (else <e>))

The value of an *if* expression is the value of a sub-expression.
Logical forms may only evaluate some sub-expressions.

- **If** expression: `(if <predicate> <consequent> <alternative>)`
- **And** and **or**: `(and <e₁> ... <eₙ>), (or <e₁> ... <eₙ>)`
- **Cond expr'n**: `(cond (<p₁> <e₁>) ... (<pₙ> <eₙ>) (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**: 
  \[(\text{and } <e_1> ... <e_n>), \quad (\text{or } <e_1> ... <e_n>)\]

- **Cond expr’n**: 
  \[(\text{cond } (<p_1> <e_1>) ... (<p_n> <e_n>) \text{ (else } <e>))\]

The value of an **if** expression is the value of a sub-expression.

- Evaluate the predicate.
- Choose a sub-expression: **<consequent>** or **<alternative>**.
Logical Special Forms

Logical forms may only evaluate some sub-expressions.

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

• **And and or:**  
  \[(\\text{and} \ <\text{e}_1> \ ... \ <\text{e}_n>), \quad (\text{or} \ <\text{e}_1> \ ... \ <\text{e}_n>)\]

• **Cond expr'n:**  
  \[(\text{cond} \ (<\text{p}_1> \ <\text{e}_1>) \ ... \ (<\text{p}_n> \ <\text{e}_n>) \ (\text{else} \ <\text{e}>))\]

The value of an **if** expression is the value of a sub-expression.

• Evaluate the predicate.

• Choose a sub-expression:  \(<\text{consequent}>\) or \(<\text{alternative}>\).

• Evaluate that sub-expression in place of the whole expression.
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 expr'n**:  `(cond (<p₁> <e₁>) ... (<pₙ> <eₙ>) (else <e>))`

The value of an **if** expression is the value of a sub-expression.

1. Evaluate the predicate.
2. Choose a sub-expression: `<consequent>` or `<alternative>`.  
3. Evaluate that sub-expression in place 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** expr'n: \( (\text{cond } (<p_1> <e_1>) ... (<p_n> <e_n>) \text{ (else } <e>)) \)

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 in place 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**: \((\text{and } \text{<e}_1 \ldots \text{<e}_n)), \quad (\text{or } \text{<e}_1 \ldots \text{<e}_n))\)
- **Cond** expr'n: \((\text{cond } (\text{<p}_1 \text{<e}_1) \ldots (\text{<p}_n \text{<e}_n}) \text{(else } \text{<e}>))\)

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 in place of the whole expression.

**Demo**
Quotation
Quotation

The *quote* special form evaluates to the quoted expression.
Quotation

The **quote** special form evaluates to the quoted expression.

```
(quote <expression>)
```
The **quote** special form evaluates to the quoted expression.

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

Evaluates to the \(<\text{expression}>\) itself, not its value!
Quotation

The **quote** special form evaluates to the quoted expression.

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

Evaluates to the `<expression>` itself, not its value!

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

The **quote** special form evaluates to the quoted expression.

**(quote <expression>)**

Evaluates to the `<expression>` itself, not its value!

'&lt;expression&gt; is shorthand for `(quote <expression>)`.

`(quote (1 2))`
Quotation

The **quote** special form evaluates to the quoted expression.

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

Evaluates to the \langle expression \rangle itself, not its value!

\'
\langle expression \rangle
\'

is shorthand for \(\text{(quote } \langle \text{expression} \rangle)\).

\[
(\text{quote (1 2)})
\]

\'(1 2)
Quotation

The **quote** special form evaluates to the quoted expression.

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

Evaluates to the `<expression>` itself, not its value!

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

\[(\text{quote } (1 2))\]

'(1 2)

The scheme_read parser converts shorthand to a combination.
Quotation

The **quote** special form evaluates to the quoted expression.

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

Evaluates to the `<expression>` itself, not its value!

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

\[(quote \ (1\ 2))\]

'`(1\ 2)`

The `scheme_read` parser converts shorthand to a combination.

Demo
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.

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

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

Lambda expressions evaluate to user-defined procedures.

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

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

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

Lambda expressions evaluate to user-defined procedures.

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

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

class LambdaProcedure(object):

    def __init__(self, formals, body, env):
        self.formals = formals
        self.body = body
        self.env = env

A scheme list of symbols
Lambda Expressions

Lambda expressions evaluate to user-defined procedures.

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

\( \text{lambda} \left( x \right) \left( \ast \ x \ x \right) \)

class LambdaProcedure(object):

def __init__(self, formals, body, env):
    self.formals = formals
    self.body = body
    self.env = env

A scheme list of symbols
A scheme expression
Lambda Expressions

Lambda expressions evaluate to user-defined procedures.

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

(lambda (x) (* x x))

class LambdaProcedure(object):
    def __init__(self, formals, body, env):
        self.formals = formals  # A scheme list of symbols
        self.body = body        # A scheme expression
        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 \texttt{lookup} and \texttt{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 \texttt{lookup} and \texttt{define}.

In Project 4, Frames do not hold return values.

\[
\begin{array}{|c|}
\hline
\text{g: Global frame} \\
\text{y} & 3 \\
\text{z} & 5 \\
\hline
\end{array}
\]
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
```

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

```
[parent=g]
```

```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>2</td>
</tr>
<tr>
<td>z</td>
<td>4</td>
</tr>
</tbody>
</table>
```

Demo
Define Expressions
Define Expressions

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

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

(define <name> <expression>)
Define Expressions

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

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

Evaluate the \langle\text{expression}\rangle.
Define Expressions

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

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

Evaluate the \text{expression}.

Bind \text{name} to the result (\text{define} method of the current frame).
Define Expressions

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

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

Evaluate the \textit{expression}.

Bind \textit{name} to the result (\textbf{define} method of the current frame).

\[(\text{define } x 2)\]
Define Expressions

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

(expression)

Evaluate the <expression>.

Bind <name> to the result (define method of the current frame).

(define x 2)

Procedure definition is a combination of define and lambda.
Define Expressions

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

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

Evaluate the \langle\text{expression}\rangle.

Bind \langle\text{name}\rangle to the result (\text{define} method of the current frame).

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

Procedure definition is a combination of define and lambda.

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

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

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

Evaluate the \(<\text{expression}>\).

Bind \(<\text{name}>\) to the result (\text{define} method of the current frame).

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

Procedure definition is a combination of define and lambda.

\[(\text{define} \ (<\text{name}> \ (<\text{formal\ parameters}>)) \ (<\text{body}>))] \\
\%(define \ <\text{name}> \ (\text{lambda} \ (<\text{formal\ parameters}>)) \ (<\text{body}>)%\)
Applying User-Defined Procedures
Applying User-Defined Procedures

Create a new frame in which formal parameters are bound to argument values, whose parent is the env of the procedure.
Applying User-Defined Procedures

Create a new frame in which formal parameters are bound to argument values, whose parent is the env of the procedure.

Evaluate the body of the procedure in the environment that starts with this new frame.
Applying User-Defined Procedures

Create a new frame in which formal parameters are bound to argument values, whose parent is the env of the procedure.

Evaluate the body of the procedure in the environment that starts with this new frame.

\[(\text{define } (f \ s) \ (\text{if } (\text{null? } s) \ '(3) \ (\text{cons} \ (\text{car} \ s) \ (f \ (\text{cdr} \ s)))))))\]
Applying User-Defined Procedures

Create a new frame in which formal parameters are bound to argument values, whose parent is the env of the procedure.

Evaluate the body of the procedure in the environment that starts with this new frame.

```
(define (f s) (if (null? s) '(3) (cons (car s) (f (cdr s)))))
```
Applying User-Defined Procedures

Create a new frame in which formal parameters are bound to argument values, whose parent is the env of the procedure.

Evaluate the body of the procedure in the environment that starts with this new frame.

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

\[(f \ (\text{list } 1 \ 2))\]
Applying User-Defined Procedures

Create a new frame in which formal parameters are bound to argument values, whose parent is the \texttt{env} of the procedure.

Evaluate the body of the procedure in the environment that starts with this new frame.

\begin{verbatim}
(define (f s) (if (null? s) '(3) (cons (car s) (f (cdr s)))))

(f (list 1 2))
\end{verbatim}
Applying User-Defined Procedures

Create a new frame in which formal parameters are bound to argument values, whose parent is the env of the procedure.

Evaluate the body of the procedure in the environment that starts with this new frame.

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

(f (list 1 2))
Applying User-Defined Procedures

Create a new frame in which formal parameters are bound to argument values, whose parent is the env of the procedure.

Evaluate the body of the procedure in the environment that starts with this new frame.

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

\[
(f \ (\text{list 1 2}))
\]
Applying User-Defined Procedures

Create a new frame in which formal parameters are bound to argument values, whose parent is the env of the procedure.

Evaluate the body of the procedure in the environment that starts with this new frame.

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

(f (list 1 2))
Applying User-Defined Procedures

Create a new frame in which formal parameters are bound to argument values, whose parent is the `env` of the procedure.

Evaluate the body of the procedure in the environment that starts with this new frame.

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

```
(f (list 1 2))
```

Demo
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[cadr[e];a];  
  T ← apply[car[e];evlis[cadr[e];a];a]];  
  T ← apply[car[e];evlis[cadr[e];a];a]]
Dynamic Scope
Dynamic Scope

The way in which names are looked up in Scheme and Python is called *lexical scope* (or *static scope*).
Dynamic Scope

The way in which names are looked up in Scheme and Python is called \textit{lexical scope} (or \textit{static scope}).

\textbf{Lexical scope:} The parent of a frame is the environment in which a procedure was \textit{defined}. 
Dynamic Scope

The way in which names are looked up in Scheme and Python is called *lexical scope* (or *static scope*).

**Lexical scope:** The parent of a frame is the environment in which a procedure was *defined*.

**Dynamic scope:** The parent of a frame is the environment in which a procedure was *called*. 
Dynamic Scope

The way in which names are looked up in Scheme and Python is called *lexical scope* (or *static scope*).

**Lexical scope:** The parent of a frame is the environment in which a procedure was *defined*.

**Dynamic scope:** The parent of a frame is the environment in which a procedure was *called*.

```
(define f (lambda (x) (+ x y)))
```
Dynamic Scope

The way in which names are looked up in Scheme and Python is called **lexical scope** (or **static scope**).

**Lexical scope:** The parent of a frame is the environment in which a procedure was *defined*.

**Dynamic scope:** The parent of a frame is the environment in which a procedure was *called*.

```
(define f (lambda (x) (+ x y)))
(define g (lambda (x y) (f (+ x x))))
```
Dynamic Scope

The way in which names are looked up in Scheme and Python is called *lexical scope* (or *static scope*).

**Lexical scope:** The parent of a frame is the environment in which a procedure was *defined*.

**Dynamic scope:** The parent of a frame is the environment in which a procedure was *called*.

```
(define f (lambda (x) (+ x y)))
(define g (lambda (x y) (f (+ x x))))
(g 3 7)
```
The way in which names are looked up in Scheme and Python is called *lexical scope* (or *static scope*).

**Lexical scope:** The parent of a frame is the environment in which a procedure was *defined*.

**Dynamic scope:** The parent of a frame is the environment in which a procedure was *called*.

```
(define f (lambda (x) (+ x y)))
(define g (lambda (x y) (f (+ x x))))
(g 3 7)
```

**Lexical scope:** The parent for f's frame is the global frame.
The way in which names are looked up in Scheme and Python is called \textit{lexical scope} (or \textit{static scope}).

\textbf{Lexical scope:} \ The parent of a frame is the environment in which a procedure was \textit{defined}.

\textbf{Dynamic scope:} \ The parent of a frame is the environment in which a procedure was \textit{called}.

\begin{verbatim}
(define f (lambda (x) (+ x y)))
(define g (lambda (x y) (f (+ x x))))
(g 3 7)
\end{verbatim}

\textbf{Lexical scope:} \ The parent for f's frame is the global frame.

\textbf{Dynamic scope:} \ The parent for f's frame is g's frame.
Dynamic Scope

The way in which names are looked up in Scheme and Python is called *lexical scope* (or *static scope*).

**Lexical scope**: The parent of a frame is the environment in which a procedure was *defined*.

**Dynamic scope**: The parent of a frame is the environment in which a procedure was *called*.

```
(define f (lambda (x) (+ x y)))
(define g (lambda (x y) (f (+ x x))))
(g 3 7)
```

**Lexical scope**: The parent for f's frame is the global frame. *Error: unknown identifier: y*

**Dynamic scope**: The parent for f's frame is g's frame.
Dynamic Scope

The way in which names are looked up in Scheme and Python is called \textit{lexical scope} (or \textit{static scope}).

\textbf{Lexical scope:} The parent of a frame is the environment in which a procedure was \textit{defined}.

\textbf{Dynamic scope:} The parent of a frame is the environment in which a procedure was \textit{called}.

\begin{verbatim}
(define f (lambda (x) (+ x y)))
(define g (lambda (x y) (f (+ x x))))
(g 3 7)
\end{verbatim}

\textbf{Lexical scope:} The parent for f's frame is the global frame. \textit{Error: unknown identifier: y}

\textbf{Dynamic scope:} The parent for f's frame is g's frame.
Dynamic Scope

The way in which names are looked up in Scheme and Python is called *lexical scope* (or *static scope*).

**Lexical scope:** The parent of a frame is the environment in which a procedure was *defined*.

**Dynamic scope:** The parent of a frame is the environment in which a procedure was *called*.

```scheme
(define f (lambda (x) (+ x y)))
(define g (lambda (x y) (f (+ x x))))
(g 3 7)
```

**Lexical scope:** The parent for f's frame is the global frame. *Error: unknown identifier: y*

**Dynamic scope:** The parent for f's frame is g's frame.

Special form to create dynamically scoped procedures

```
(define f (lambda (x) (+ x y)))
(define g (lambda (x y) (f (+ x x))))
(g 3 7)
```