Announcements

- HW10 deadline extended to 11:59pm Thursday

- Scheme project out
Read-Eval-Print Loop
Read-Eval-Print Loop

The user interface to many programming languages is an interactive loop, which
The user interface to many programming languages is an interactive loop, which

• Reads an expression from the user,
The user interface to many programming languages is an interactive loop, which
• Reads an expression from the user,
• Parses the input to build an expression tree,
Read-Eval-Print Loop

The user interface to many programming languages is an interactive loop, which

• Reads an expression from the user,
• Parses the input to build an expression tree,
• Evaluates the expression tree,
Read-Eval-Print Loop

The user interface to many programming languages is an interactive loop, which

• Reads an expression from the user,
• Parses the input to build an expression tree,
• Evaluates the expression tree,
• Prints the resulting value of the expression
Read-Eval-Print Loop

The user interface to many programming languages is an interactive loop, which

• Reads an expression from the user,
• Parses the input to build an expression tree,
• Evaluates the expression tree,
• Prints the resulting value of the expression

The REPL handles errors by printing informative messages for the user, rather than crashing
The user interface to many programming languages is an interactive loop, which

• Reads an expression from the user,
• Parses the input to build an expression tree,
• Evaluates the expression tree,
• Prints the resulting value of the expression

The REPL handles errors by printing informative messages for the user, rather than crashing

A well-designed REPL should not crash on any input!
The Structure of an Evaluator
The Structure of an Evaluator

Eval

Apply
The Structure of an Evaluator

Base cases:

Eval

Apply
The Structure of an Evaluator

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

Base cases:
- Primitive values (numbers)

Recursive calls:
The Structure of an Evaluator

Base cases:
• Primitive values (numbers)

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

Base cases:
• Primitive values (numbers)

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

Base cases:
- Primitive values (numbers)

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

Base cases:
- Primitive values (numbers)

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

Base cases:
- Built-in primitive procedures
The Structure of an Evaluator

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 Evaluator

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 Evaluator

**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:
The Structure of an Evaluator

**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
The Structure of an Evaluator

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

Recursive calls:
- Eval(body) of user-defined proc's
The Structure of an Evaluator

**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 Evaluator

**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:
The `scheme_eval` function dispatches on expression form:

- Symbols are bound to values in the current environment
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
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
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>)
```
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>)
```
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>)
```
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

\[
\begin{align*}
(\text{if } <\text{predicate}> & \; <\text{consequent}> & \; <\text{alternative}>) \\
(\text{lambda } (<\text{formal-parameters}>) & \; <\text{body}> ) \\
(\text{define } & \; <\text{name}> \; <\text{expression}> ) \\
(<\text{operator}> & \; <\text{operand} \; 0> \; ... \; <\text{operand} \; k> )
\end{align*}
\]
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.
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.
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>)
(<operator> <operand 0> ... <operand k>)
```

Special forms are identified by the first list element.
Anything not a known special form is a call expression.
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)))))
```
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

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

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

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

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

```
(define (f s) (if (null? s) '(3) (cons (car s) (f (cdr s)))))
(f (list 1 2))
```
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 forms may only evaluate some sub-expressions.

- **If** expression: `(if <predicate> <consequent> <alternative>)`
- **And** and **or**: `(and <e₁> ... <eₙ>), (or <e₁> ... <eₙ>)`
Logical Special Forms

Logical forms may only evaluate some sub-expressions.

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

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

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

- Evaluate the predicate.

- Choose a sub-expression: `<consequent>` or `<alternative>`
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.

- 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: \(<\text{consequent}>\) 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.

- Evaluate the predicate.
- Choose a sub-expression: `<consequent>` or `<alternative>`
- Evaluate that sub-expression in place of the whole expression.
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

\[(quote<expression>)\]

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

The `quote` special form evaluates to the quoted expression

```
(quote <expression>)
```

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

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

`(quote (1 2))`
The *quote* special form evaluates to the quoted expression

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

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

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

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

' (1 2)
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>)**

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

'**(1 2)**'  

The **scheme_read** parser converts shorthand to a combination
Lambda Expressions
Lambda Expressions

Lambda expressions evaluate to user-defined procedures
Lambda Expressions

Lambda expressions evaluate to user-defined procedures

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

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

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

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  # 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) \ (* \ 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
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  # 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 `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
```
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

[parent=g]
  x | 2
  z | 4
```
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

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

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

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

Evaluate the `<expression>`
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 (define method of the current Frame)
Define Expressions

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

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

Evaluate the \textbf{expression}

Bind \textbf{name} to the result (\textbf{define} method of the current \textbf{Frame})

\[(\text{define } x \ 2)\]
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 (define method of the current Frame)

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

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} \ <\text{name}> \ <\text{expression}>\)]

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

Bind \(<\text{name}>\) to the result (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 Expressions

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

```
(define <name> <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 (<name> <formal parameters>) <body>)

(define <name> (lambda (<formal parameters>) <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))))\text{)}
\]
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))))
\end{verbatim}

\begin{itemize}
  \item \texttt{g}: Global frame
  \item \texttt{f}: LambdaProcedure instance [parent=\texttt{g}]
\end{itemize}
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))
```

- g: Global frame
- f: LambdaProcedure instance [parent=g]
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.

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

\[
(f \text{ (list 1 2)})
\]
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))
```
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

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

\[
(f \ (\text{list 1 2}))
\]
Eval/Apply in Lisp 1.5
Eval/Apply in Lisp 1.5

\[
\text{apply}[\text{fn}; \text{x}; \text{a}] = \\
\begin{align*}
\text{atom}[\text{fn}] & \rightarrow [\text{eq}[\text{fn}; \text{CAR}] \rightarrow \text{caar}[\text{x}]; \\
& \quad \text{eq}[\text{fn}; \text{CDR}] \rightarrow \text{cdar}[\text{x}]; \\
& \quad \text{eq}[\text{fn}; \text{CONS}] \rightarrow \text{cons}[\text{car}[\text{x}]; \text{cdadr}[\text{x}]]; \\
& \quad \text{eq}[\text{fn}; \text{ATOM}] \rightarrow \text{atom}[\text{car}[\text{x}]]; \\
& \quad \text{eq}[\text{fn}; \text{EQ}] \rightarrow \text{eq}[\text{car}[\text{x}]; \text{cdadr}[\text{x}]]; \\
& \quad \text{T} \rightarrow \text{apply}[\text{eval}[\text{fn}; \text{a}]; \text{x}; \text{a}]]; \\
\text{eq}[\text{car}[\text{fn}]; \text{LAMBDA}] & \rightarrow \text{eval}[\text{caddr}[\text{fn}]; \text{pairlis}[\text{cadr}[\text{fn}]; \text{x}; \text{a}]]; \\
\text{eq}[\text{car}[\text{fn}]; \text{LABEL}] & \rightarrow \text{apply}[\text{caddr}[\text{fn}]; \text{x}; \text{cons}[\text{cons}[\text{cadr}[\text{fn}]; \\
& \quad \text{caddr}[\text{fn}]; \text{a}]]]
\end{align*}
\]

\[
\text{eval}[\text{e}; \text{a}] = [\text{atom}[\text{e}] \rightarrow \text{cdr}[\text{assoc}[\text{e}; \text{a}]]; \\
\text{atom}[\text{car}[\text{e}]] \rightarrow \\
\begin{align*}
& [\text{eq}[\text{car}[\text{e}]; \text{QUOTE}] \rightarrow \text{cadr}[\text{e}]; \\
& \quad \text{eq}[\text{car}[\text{e}]; \text{COND}] \rightarrow \text{evcon}[\text{cdr}[\text{e}]; \text{a}]; \\
& \quad \text{T} \rightarrow \text{apply}[\text{car}[\text{e}]; \text{evlis}[\text{cdr}[\text{e}]; \text{a}]; \text{a}]]; \\
& \quad \text{T} \rightarrow \text{apply}[\text{car}[\text{e}]; \text{evlis}[\text{cdr}[\text{e}]; \text{a}]; \text{a}]]
\end{align*}
\]
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 *lexical scope* (or *static scope*)

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

\[
\text{(define f (lambda (x) (+ x y)))}
\]
\[
\text{(define g (lambda (x y) (f (+ x x)))))}
\]
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)
```
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
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

**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 *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

13
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

13