Structure and Interpretation of Computer Programs

COMP200
TODAY
Interpretation

- Parts of an interpreter
- Arithmetic calculator
- Names
- Conditionals and if
- Store procedures in the environment
- Environment as explicit parameter
- Defining new procedures
TODAY

Why do we need an Interpreter?
Why do we need an Interpreter?

- Abstractions let us bury details and focus on use of modules to solve large systems.
TODAY

Why do we need an Interpreter?

• Abstractions let us bury details and focus on use of modules to solve large systems.

• Need to unwind abstractions at execution time to deduce meaning
TODAY

Why do we need an Interpreter?

• Abstractions let us bury details and focus on use of modules to solve large systems.
• Need to unwind abstractions at execution time to deduce meaning
• Have seen such a process - Environment Model
• Now want to describe that process as a procedure
INTERPRETER

Stages

Lexical Analyzer

Parser

Evaluator

Environment

Printer
INTERPRETER
Input-Output of Stages

Lexical Analyzer

Parser

Evaluator

Environment

Printer

“(average 4 (+ 5 5))”
INTERPRETER
Input-Output of Stages

Lexical Analyzer

Parser

Evaluator

Environment

Printer

"(average 4 (+ 5 5))"

( average 4 ( + 5 5 ) )
INTERPRETER

Input-Output of Stages

Lexical Analyzer

Parser

Evaluator

Environment

Printer

“(average 4 (+ 5 5))”
INTERPRETER
Input-Output of Stages

Lexical Analyzer

Parser

Evaluator

Environment

Printer

“(average 4 (+ 5 5))”

4

symbol +

5

7
The input string is "(average 4 (+ 5 5))". It is processed through the stages of the interpreter:

1. **Lexical Analyzer** extracts tokens from the input.
2. **Parser** constructs an abstract syntax tree (AST) from the tokens.
3. **Evaluator** traverses the AST and performs the computation.
4. **Environment** provides the necessary context for symbol resolution.
5. **Printer** outputs the result.

The AST of "(average 4 (+ 5 5))" is shown with symbols and operations: `+`, `average`, and a symbol for the result. The evaluation process is depicted with arrows indicating the flow of operation and the computation steps, leading to the final result of 7.
"(average 4 (+ 5 5))"

```
( average 4 (+ 5 5) )
```

```
symbol average

4

symbol +

5

5

"7"

Input-Output of Stages
1. **Lexical analyzer**
   - break up input string into "words" called tokens

2. **Parser**
   - convert linear sequence of tokens to a tree
   - like diagramming sentences in elementary school
   - also convert self-evaluating tokens to their internal values
     - `#f` is converted to the internal false value
3. Evaluator
   • follow language rules to convert parse tree to a value
   • read and modify the environment as needed

4. Printer
   • convert value to human-readable output string
INTERPRETER

Our Goal

• Implement an interpreter for a programming language
Our Goal

• Implement an interpreter for a programming language

• Only write **evaluator** and **environment**
  • use scheme's **reader** for lexical analysis and parsing
  • use scheme's **printer** for output
  • to do this, our language must look like scheme
INTERPRETER
Our Goal

• Implement an interpreter for a programming language
• Only write evaluator and environment
  • use scheme's reader for lexical analysis and parsing
  • use scheme's printer for output
  • to do this, our language must look like scheme
• Call the language scheme*
  • All names end with a star
Our Goal

- Implement an interpreter for a programming language
- Only write evaluator and environment
  - use scheme's reader for lexical analysis and parsing
  - use scheme's printer for output
  - to do this, our language must look like scheme
- Call the language scheme*
  - All names end with a star
- Start with a simple calculator for arithmetic
- Progressively add scheme* features
1. ARITHMETIC CALCULATOR

Start Simple

Evaluate arithmetic expressions of two arguments:

(plus* 24 (plus* 5 6))
1. ARITHMETIC CALCULATOR

Code
1. ARITHMETIC CALCULATOR

Walking through a Tree

(plus* 24 (plus* 5 6))
1. ARITHMETIC CALCULATOR

Walking through a Tree

\[(\text{plus* } 24 \text{ (plus* } 5 \text{ 6)})\]
1. ARITHMETIC CALCULATOR

Walking through a Tree

(sum? checks the tag)

(plus* 24 (plus* 5 6))
1. ARITHMETIC CALCULATOR

Walking through a Tree

\[(\text{eval-sum}) \quad (\text{plus* 24 (plus* 5 6))}\]
1. ARITHMETIC CALCULATOR

Walking through a Tree

\((\text{eval-sum} \rightarrow \text{plus}* \rightarrow 24 \rightarrow \text{plus}* \rightarrow 5 \rightarrow 6)\)

\((+ (\text{eval} 24) (\text{eval} \rightarrow \text{plus}* \rightarrow 5 \rightarrow 6))\)
1. ARITHMETIC CALCULATOR

Walking through a Tree

\[
\text{(eval-sum} \rightarrow 24 \rightarrow \text{plus*} \rightarrow 5 \rightarrow 6 \rightarrow \text{plus*} \rightarrow (\text{eval 24}) \rightarrow \text{eval 5} \rightarrow \text{eval 6})
\]

\[
(\text{plus* 24 (plus* 5 6))}
\]
1. ARITHMETIC CALCULATOR

Walking through a Tree

\[(\text{eval-sum} \rightarrow \text{plus*} \rightarrow 24) \rightarrow \text{plus*} \rightarrow (5 \rightarrow 6) )\]

\[ (+ (\text{eval 24}) \rightarrow (\text{eval} \rightarrow \text{plus*} \rightarrow 5 \rightarrow 6) ) )\]

\[ (+ (\text{eval 5}) \rightarrow (\text{eval 6}) ) \]
1. ARITHMETIC CALCULATOR

Start Simple

(plus* 24 (plus* 5 6))

What are the argument and return values of eval each time it is called in the evaluation?
1. ARITHMETIC CALCULATOR

Start Simple

(plus* 24 (plus* 5 6))

What are the argument and return values of \texttt{eval} each time it is called in the evaluation?

\texttt{(eval '(plus* 24 (plus* 5 6)))}
1. ARITHMETIC CALCULATOR

Start Simple

(plus* 24 (plus* 5 6))

What are the argument and return values of eval each time it is called in the evaluation?

| (eval-sum '(plus* 24 (plus* 5 6))) | (eval '(plus* 24 (plus* 5 6))) |
1. ARITHMETIC CALCULATOR

Start Simple

(plus* 24 (plus* 5 6))

What are the argument and return values of eval each time it is called in the evaluation?

<table>
<thead>
<tr>
<th>(eval 24)</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td>(eval-sum '(plus* 24 (plus* 5 6)))</td>
<td></td>
</tr>
<tr>
<td>(eval '(plus* 24 (plus* 5 6)))</td>
<td></td>
</tr>
</tbody>
</table>
1. ARITHMETIC CALCULATOR

Start Simple

(plus* 24 (plus* 5 6))

What are the argument and return values of `eval` each time it is called in the evaluation?

<table>
<thead>
<tr>
<th>(eval 24)</th>
<th>24</th>
<th>(eval '(plus* 5 6))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(eval-sum '(plus* 24 (plus* 5 6)))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(eval '(plus* 24 (plus* 5 6)))</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
1. ARITHMETIC CALCULATOR

Start Simple

\[(\text{plus}^* 24 (\text{plus}^* 5 6))\]

What are the argument and return values of \text{eval} each time it is called in the evaluation?

<table>
<thead>
<tr>
<th>(eval 24)</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td>(eval-sum '(plus* 5 6))</td>
<td>(eval '(plus* 5 6))</td>
</tr>
<tr>
<td>(eval-sum '(plus* 24 (plus* 5 6)))</td>
<td>(eval '(plus* 24 (plus* 5 6)))</td>
</tr>
</tbody>
</table>
1. ARITHMETIC CALCULATOR

Start Simple

`(plus* 24 (plus* 5 6))`

What are the argument and return values of `eval` each time it is called in the evaluation?

<table>
<thead>
<tr>
<th></th>
<th>24</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>eval 24</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>eval 5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>eval 6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>eval-sum ' (plus* 5 6)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>eval ' (plus* 5 6)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>eval-sum ' (plus* 24 (plus* 5 6))</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>eval ' (plus* 24 (plus* 5 6))</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
1. ARITHMETIC CALCULATOR

Start Simple

\[(\text{plus* } 24 \ (\text{plus* } 5 \ 6))\]

What are the argument and return values of `eval` each time it is called in the evaluation?

\[
\begin{array}{c|c|c}
\text{(eval 24)} & 24 & \text{(eval 5)} & 5 & \text{(eval 6)} & 6 \\
\hline
\text{(eval-sum '(plus* 5 6))} & \text{(eval '(plus* 5 6))} & 11 & 11 \\
\text{(eval-sum '(plus* 24 (plus* 5 6)))} & \text{(eval '(plus* 24 (plus* 5 6)))} & 35 & 35 \\
\end{array}
\]
1. ARITHMETIC CALCULATOR

Things to Observe

- **cond** determines the expression type

- no work to do on numbers
  - scheme's reader has already done the work
  - it converts a sequence of characters like "24" to an internal binary representation of the number 24

- **eval-sum** recursively calls **eval** on both argument expressions
2. NAMES

• Extend the calculator to store intermediate results as named values

    (define* x* (plus* 4 5))    Store result as x*
    (plus* x* 2)                Use that result
2. NAMES

- Extend the calculator to store intermediate results as named values
  
  \[
  (\text{define* } x* (\text{plus*} 4 5)) \quad \text{Store result as } x*
  \]
  \[
  (\text{plus* } x* 2) \quad \text{Use that result}
  \]

- Store bindings between names and values in a table
2. NAMES

Code
2. NAMES

\texttt{define}

\texttt{(eval \ '(define* \ x* \ (plus* 4 5)))}
2. NAMES

define

(eval '(define* x* (plus* 4 5)))
  (eval '(plus* 4 5))
    (eval 4) ==> 4
    (eval 5) ==> 5
  ==> 9
2. NAMES

define

(eval ' (define* x* (plus* 4 5)))
  (eval ' (plus* 4 5))
    (eval 4) ==> 4
    (eval 5) ==> 5
  ==> 9

<table>
<thead>
<tr>
<th>name</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x*</td>
<td>9</td>
</tr>
</tbody>
</table>
2. NAMES

\[
\begin{array}{c|c}
\text{name} & \text{value} \\
\hline
x^* & 9 \\
\end{array}
\]

\[\text{(eval (define* x^* (plus* 4 5)))} \]
\[\text{(eval (plus* 4 5))} \]
\[\text{(eval 4) ===> 4} \]
\[\text{(eval 5) ===> 5} \]

==> 9

==> undefined
2. NAMES

(define* x* (plus* 4 5))
  (eval '(plus* 4 5))
    (eval 4) ==> 4
    (eval 5) ==> 5
  ==> 9
==> undefined

(eval '(plus* x* 2))
  (eval 'x*) ==> 9
  (eval 2) ==> 2
==> 11

<table>
<thead>
<tr>
<th>name</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x*</td>
<td>9</td>
</tr>
</tbody>
</table>
2. NAMES

Things to Observe

- Use scheme function `symbol?` to check for a name
  - the reader converts sequences of characters like "x*" to symbols in the parse tree
2. NAMES

Things to Observe

• Use scheme function `symbol?` to check for a name
  • the reader converts sequences of characters like "x*" to symbols in the parse tree

• Can use any implementation of the `table` ADT
2. NAMES

Things to Observe

- Use scheme function `symbol?` to check for a name
  - the reader converts sequences of characters like "x*" to symbols in the parse tree

- Can use any implementation of the `table` ADT

- `eval-define` recursively calls `eval` on the second subtree but not on the first one

- `eval-define` returns a special undefined value
3. CONDITIONALS

if

• Extend the calculator to handle conditionals and if.

(if* (greater* y* 6) (plus* y* 2) 15)
3. CONDITIONALS

- Extend the calculator to handle conditionals and if.

\[(\text{if\*} (\text{greater\*} \ y \ 6) (\text{plus\*} \ y \ 2) 15)\]

\text{greater\*: an operation that returns a boolean}
\text{if\*: an operation that evaluates the first subexp, checks if value is true or false}
3. CONDITIONALS

Code
(if* (greater* y* 6) (plus* y* 2) 15)
3. CONDITIONALS

Walking through a Tree

\[(\text{if* } (\text{greater* } y* \ 6) \ (\text{plus* } y* \ 2) \ 15)\]

\[(\text{eval } \rightarrow \begin{array}{c}
\text{greater*} \\
greater* \\
\end{array} \rightarrow \begin{array}{c}
y* \\
y* \\
\end{array} \rightarrow \begin{array}{c}
6 \\
6 \\
\end{array} )\]
3. CONDITIONALS

Walking through a Tree

(if* (greater* y* 6) (plus* y* 2) 15)

(eval greater* y* 6 ) or (eval 15 )

(eval plus* y* 2 )
3. CONDITIONALS

\[ (\text{eval }'(\text{if*} (\text{greater* }y* 6) (\text{plus* }y* 2) 15)) \]
\[ (\text{eval }'(\text{greater* }y* 6)) \]
\[ (\text{eval }'y*) \Rightarrow 9 \]
\[ (\text{eval }6) \Rightarrow 6 \]
\[ \Rightarrow \#t \]
\[ (\text{eval }'(\text{plus* }y* 2)) \]
\[ (\text{eval }'y*) \Rightarrow 9 \]
\[ (\text{eval }2) \Rightarrow 2 \]
\[ \Rightarrow 11 \]
\[ \Rightarrow 11 \]
3. CONDITIONALS

Things to Observe

- `eval-greater` is just like `eval-sum`
  - recursively call `eval` on both argument expressions
  - call scheme `>` to compute value
3. CONDITIONALS

Things to Observe

• **eval-greater** is just like **eval-sum**
  - recursively call **eval** on both argument expressions
  - call scheme > to compute value

• **eval-if** does not call **eval** on all argument expressions:
  - call **eval** on the predicate
  - call **eval** on the consequent or on the alternative but not both
4. STORE OPERATORS IN THE ENVIRONMENT

Adding New Things

- Want to add lots of operators but keep `eval` short.
• Want to add lots of operators but keep `eval` short.

• Operations like `plus*` and `greater*` are similar
  • evaluate all the argument subexpressions
  • perform the operation on the resulting values
4. STORE OPERATORS IN THE ENVIRONMENT

Adding New Things

• Want to add lots of operators but keep `eval` short.

• Operations like `plus*` and `greater*` are similar
  • evaluate all the argument subexpressions
  • perform the operation on the resulting values

• Call this standard pattern an `application`
  • Implement a single case in `eval` for all application
4. STORE OPERATORS IN THE ENVIRONMENT

Approach

- **eval** the first subexpression of an application
- put a name in the environment for each operation
- value of that name is an **procedure**
- **apply** the **procedure** to the **operands**
4. STORE OPERATORS IN THE ENVIRONMENT

Code
4. Store Operators in the Environment

<table>
<thead>
<tr>
<th>name</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>z*</td>
<td>9</td>
</tr>
<tr>
<td>true*</td>
<td>#t</td>
</tr>
<tr>
<td>greater*</td>
<td></td>
</tr>
<tr>
<td>plus*</td>
<td></td>
</tr>
</tbody>
</table>

Environment

- STORE OPERATORS IN THE ENVIRONMENT
4. STORE OPERATORS IN THE ENVIRONMENT

Evaluation

(eval '(plus* 9 6))
4. STORE OPERATORS IN THE ENVIRONMENT

Evaluation

(eval '(plus* 9 6))
(apply (eval 'plus*) (map eval '(9 6)))
(eval '(plus* 9 6))
(apply (eval 'plus*) (map eval '(9 6)))
(apply '(primitive '#[add])
  (list (eval 9) (eval 6)))

4. STORE OPERATORS IN THE ENVIRONMENT
4. STORE OPERATORS IN THE ENVIRONMENT

Evaluation

(eval '(plus* 9 6))
(apply (eval 'plus*) (map eval '(9 6)))
(apply '(primitive #[add])
    (list (eval 9) (eval 6))
(apply '(primitive #[add]) '(9 6))
(eval '(plus* 9 6))
(apply (eval 'plus*) (map eval '(9 6)))
(apply '(primitive #[add])
   (list (eval 9) (eval 6))
(apply '(primitive #[add]) '(9 6))
(scheme-apply
   (get-scheme-procedure '(primitive #[add]))
   '(9 6))
4. STORE OPERATORS IN THE ENVIRONMENT

Evaluation

(eval '(plus* 9 6))
(apply (eval 'plus*) (map eval '(9 6)))
(apply '(primitive #[add])
   (list (eval 9) (eval 6))
(apply '(primitive #[add]) '(9 6))
(scheme-apply
   (get-scheme-procedure '(primitive #[add]))
   '(9 6))
(scheme-apply #[add] '(9 6))
4. STORE OPERATORS IN THE ENVIRONMENT

**Evaluation**

(eval '(plus* 9 6))
(apply (eval 'plus*) (map eval '(9 6)))
(apply '(primitive #[add])
   (list (eval 9) (eval 6))
(apply '(primitive #[add]) '(9 6))
(scheme-apply
   (get-scheme-procedure '(primitive #[add]))
   '(9 6))
(scheme-apply #[add] '(9 6))
15
4. STORE OPERATORS IN THE ENVIRONMENT

**Evaluation**

(eval '(if* true* 10 15))

4. STORE OPERATORS IN THE ENVIRONMENT

**Evaluation**

```
(eval '(if* true* 10 15))
(eval-if '(if* true* 10 15))
```
4. STORE OPERATORS IN THE ENVIRONMENT

Evaluation

(eval '(if* true* 10 15))
(eval-if '(if* true* 10 15))
(let ((test (eval 'true*))) (cond ...))
(eval '(if* true* 10 15))
(eval-if '(if* true* 10 15))
(let ((test (eval 'true*))) (cond ...))
(let ((test (lookup 'true*))) (cond ...))

4. STORE OPERATORS IN THE ENVIRONMENT

Evaluation
4. STORE OPERATORS IN THE ENVIRONMENT

Evaluation

(eval '(if* true* 10 15))
(eval-if '(if* true* 10 15))
(let ((test (eval 'true*))) (cond ...))
(let ((test (lookup 'true*))) (cond ...))
(let ((test #t)) (cond ...))
4. STORE OPERATORS IN THE ENVIRONMENT

Evaluation

(eval '(if* true* 10 15))
(eval-if '(if* true* 10 15))
(let ((test (eval 'true*))) (cond ...))
(let ((test (lookup 'true*))) (cond ...))
(let ((test #t)) (cond ...))
(eval 10)
4. STORE OPERATORS IN THE ENVIRONMENT

**Evaluation**

```
(eval '(if* true* 10 15))
(eval-if '(if* true* 10 15))
(let ((test (eval 'true*))) (cond ...))
(let ((test (lookup 'true*))) (cond ...))
(let ((test #t)) (cond ...))
(eval 10)
10
```
4. STORE OPERATORS IN THE ENVIRONMENT

Evaluation

(eval '(if* true* 10 15))
(eval-if '(if* true* 10 15))
(let ((test (eval 'true*))) (cond ...))
(let ((test (lookup 'true*))) (cond ...))
(let ((test #t)) (cond ...))
(eval 10)
10

Apply is never called!
4. STORE OPERATORS IN THE ENVIRONMENT

Things to Observe

• Applications must be last case in \texttt{eval}
  • no tag check

• Apply is never called:
  • applications evaluate all subexpressions
  • expressions that need special handling, like \texttt{if*}, gets their own case in \texttt{eval}