Lecture 21: Interpreters I
Announcements
Roadmap

- Introduction
- Functions
- Data
- Mutability
- Objects
- Interpretation
- Paradigms
- Applications
Roadmap

- Introduction
- Functions
- Data
- Mutability
- Objects
- Interpretation
- Paradigms
- Applications

- This week (Interpretation), the goals are:
This week (Interpretation), the goals are:

- To learn a new language, Scheme, in two days!
This week (Interpretation), the goals are:

- To learn a new language, Scheme, in two days!
- To understand how interpreters work, using Scheme as an example
Programming Languages
Programming Languages

- Computers can execute programs written in many different programming languages. How?
Programming Languages

- Computers can execute programs written in many different programming languages. How?
- Computers only deal with *machine languages* (0s and 1s), where statements are direct commands to the hardware.
Programming Languages

- Computers can execute programs written in many different programming languages. How?
- Computers only deal with *machine languages* (0s and 1s), where statements are direct commands to the hardware.
- Programs written in languages like Python are *compiled*, or translated, into these machine languages.
Programming Languages

• Computers can execute programs written in many different programming languages. How?

• Computers only deal with *machine languages* (0s and 1s), where statements are direct commands to the hardware.

• Programs written in languages like Python are *compiled*, or translated, into these machine languages.

• Python programs are first compiled into Python *bytecode*, which has the benefit of being system-independent.
Programming Languages

• Computers can execute programs written in many different programming languages. How?

• Computers only deal with *machine languages* (0s and 1s), where statements are direct commands to the hardware

• Programs written in languages like Python are *compiled*, or translated, into these machine languages

• Python programs are first compiled into Python *bytecode*, which has the benefit of being system-independent

• You can look at Python bytecode using the *dis* module
Programming Languages (demo)

- Computers can execute programs written in many different programming languages. How?
- Computers only deal with machine languages (0s and 1s), where statements are direct commands to the hardware.
- Programs written in languages like Python are compiled, or translated, into these machine languages.
- Python programs are first compiled into Python bytecode, which has the benefit of being system-independent.
- You can look at Python bytecode using the dis module.
Programming Languages

- Computers can execute programs written in many different programming languages. How?
- Computers only deal with *machine languages* (0s and 1s), where statements are direct commands to the hardware.
- Programs written in languages like Python are *compiled*, or translated, into these machine languages.
- Python programs are first compiled into Python *bytecode*, which has the benefit of being system-independent.
- You can look at Python bytecode using the `dis` module.

Python 3 Bytecode

```
def square(x):
    return x * x
```

```
LOAD_FAST 0 (x)
LOAD_FAST 0 (x)
BINARY_MULTIPLY
RETURN_VALUE
```

```
from dis import dis
dis(square)
```
Interpretation
Interpretation

- Compilers are complicated, and the topic of future courses
Interpretation

- Compilers are complicated, and the topic of future courses.
- In this course, we will focus on interpreters, programs that execute other programs written in a particular language.
Interpretation

- Compilers are complicated, and the topic of future courses.
- In this course, we will focus on **interpreters**, programs that execute other programs written in a particular language.
- The Python interpreter is a program written in C.
Interpretation

- Compilers are complicated, and the topic of future courses
- In this course, we will focus on *interpreters*, programs that execute other programs written in a particular language
- The Python interpreter is a program written in C
  - After compiling it to machine code, it can be run to interpret Python programs
Interpretation

• Compilers are complicated, and the topic of future courses
• In this course, we will focus on interpreters, programs that execute other programs written in a particular language
• The Python interpreter is a program written in C
  • After compiling it to machine code, it can be run to interpret Python programs
• The last project in this course is to write a Scheme interpreter in Python
Interpretation

• Compilers are complicated, and the topic of future courses

• In this course, we will focus on *interpreters*, programs that execute other programs written in a particular language

• The Python interpreter is a program written in C
  • After compiling it to machine code, it can be run to interpret Python programs

• The last project in this course is to write a Scheme interpreter in Python
  • The Scheme interpreter can then be run using the Python interpreter to interpret Scheme programs
Interpretation

- Compilers are complicated, and the topic of future courses.
- In this course, we will focus on *interpreters*, programs that execute other programs written in a particular language.
- The Python interpreter is a program written in C.
  - After compiling it to machine code, it can be run to interpret Python programs.
- The last project in this course is to write a Scheme interpreter in Python.
  - The Scheme interpreter can then be run using the Python interpreter to interpret Scheme programs.
- To create a new programming language, we either need: 
Interpretation

• Compilers are complicated, and the topic of future courses

• In this course, we will focus on *interpreters*, programs that execute other programs written in a particular language

• The Python interpreter is a program written in C
  • After compiling it to machine code, it can be run to interpret Python programs

• The last project in this course is to write a Scheme interpreter in Python
  • The Scheme interpreter can then be run using the Python interpreter to interpret Scheme programs

• To create a new programming language, we either need a:
  • *Specification* of the syntax and semantics of the language
Interpretation

• Compilers are complicated, and the topic of future courses
• In this course, we will focus on interpreters, programs that execute other programs written in a particular language
• The Python interpreter is a program written in C
  • After compiling it to machine code, it can be run to interpret Python programs
• The last project in this course is to write a Scheme interpreter in Python
  • The Scheme interpreter can then be run using the Python interpreter to interpret Scheme programs
• To create a new programming language, we either need a:
  • Specification of the syntax and semantics of the language
  • Canonical implementation of either a compiler or interpreter for the language
The Scheme Interpreter
The Scheme Interpreter

- An interpreter for Scheme must take in text (Scheme code) as input and output the values from interpreting the text.
The Scheme Interpreter

- An interpreter for Scheme must take in text (Scheme code) as input and output the values from interpreting the text.
The Scheme Interpreter

- An interpreter for Scheme must take in text (Scheme code) as input and output the values from interpreting the text.
The Scheme Interpreter

- An interpreter for Scheme must take in text (Scheme code) as input and output the values from interpreting the text.
The Scheme Interpreter

- An interpreter for Scheme must take in text (Scheme code) as input and output the values from interpreting the text.
The Scheme Interpreter

- An interpreter for Scheme must take in text (Scheme code) as input and output the values from interpreting the text.

- The job of the parser is to take in text and perform *syntactic analysis* to convert it into expressions that the evaluator can understand.
The Scheme Interpreter

• An interpreter for Scheme must take in text (Scheme code) as input and output the values from interpreting the text.

• The job of the parser is to take in text and perform **syntactic analysis** to convert it into expressions that the evaluator can understand.

• The job of the evaluator is to read in expressions and perform **semantic analysis** to evaluate the expressions and output the corresponding values.
Calculator
Calculator

• Building an interpreter for a language is a lot of work
Calculator

- Building an interpreter for a language is a lot of work
- Today, we’ll build an interpreter for a subset of Scheme
Calculator

• Building an interpreter for a language is a lot of work
• Today, we’ll build an interpreter for a subset of Scheme
  • We will support +, -, *, /, integers, and floats
Calculator

- Building an interpreter for a language is a lot of work
- Today, we’ll build an interpreter for a subset of Scheme
  - We will support +, -, *, /, integers, and floats
- We will call this simple language Calculator
Calculator

- Building an interpreter for a language is a lot of work
- Today, we’ll build an interpreter for a subset of Scheme
  - We will support +, -, *, /, integers, and floats
- We will call this simple language Calculator
- In lab, discussion, and next lecture, we will look at more complicated examples
Building an interpreter for a language is a lot of work

Today, we’ll build an interpreter for a subset of Scheme
  • We will support +, -, *, /, integers, and floats
  • We will call this simple language Calculator

In lab, discussion, and next lecture, we will look at more complicated examples
• Building an interpreter for a language is a lot of work
• Today, we’ll build an interpreter for a subset of Scheme
  • We will support +, -, *, /, integers, and floats
• We will call this simple language Calculator
• In lab, discussion, and next lecture, we will look at more complicated examples

```
calc> (/ (+ 8 7) 5)
3.0
```
• Building an interpreter for a language is a lot of work
• Today, we’ll build an interpreter for a subset of Scheme
  • We will support +, -, *, /, integers, and floats
• We will call this simple language Calculator
• In lab, discussion, and next lecture, we will look at more complicated examples

calc> (/ (+ 8 7) 5)
3.0
calc> (+ (* 3
  (+ (* 2 4)
    (+ 3 5)))
  (+ (- 10 7)
    6))
57
Parsing

From text to expressions
Parsing
 Parsing

- The parser converts text into expressions
The parser converts text into expressions
The parser converts text into expressions
Parsing

- The parser converts text into expressions
Parsing

- The parser converts text into expressions

Text → Lexical Analysis → Tokens → Syntactic Analysis → Expressions
Parsing

- The parser converts text into expressions

\[ (+ 1) \]
The parser converts text into expressions.

Text → Lexical Analysis → Tokens → Syntactic Analysis → Expressions

'(+ 1'
' (- 23)'
The parser converts text into expressions.

Expressions:

- '(+ 1)
- '(- 23)
- '(* 4 5.6))'
 Parsing

• The parser converts text into expressions

Text

Lexical Analysis

Tokens

Syntactic Analysis

Expressions

'(+ 1'
'   (- 23)'  
'   (* 4 5.6))'
The parser converts text into expressions.
The parser converts text into expressions.
Parsing

- The parser converts text into expressions

```
(+ 1
   (- 23)
   (* 4 5.6))

'(+ 1'
'   (- 23)
'   (* 4 5.6))'

[', '+', 1
'   ', '−', 23, ']
'   ', '*', 4, 5.6, ']', ')')
```
Parsing

• The parser converts text into expressions

Lexical Analysis

Tokens

Syntactic Analysis

Expressions

• Iterative process

'(+ 1'
'  (- 23)'
'  (* 4 5.6))'

[',( '+', 1
'  (', '-', 23, ')'n'  (', '*', 4, 5.6, ')', ')')']
Parsing

- The parser converts text into expressions

- Iterative process
- Checks number of parentheses
The parser converts text into expressions

- Iterative process
- Checks number of parentheses
- Checks for malformed tokens
Parsing

- The parser converts text into expressions

---

**Lexical Analysis**

Text

```
(+ 1)

(- 23)

(* 4 5.6))
```

**Tokens**

```
['(', '+', 1]

['(', '-', 23, ']

['(', '*', 4, 5.6, ')', ']
```

**Syntactic Analysis**

**Expressions**

---

- Iterative process
- Checks number of parentheses
- Checks for malformed tokens
- Determines types of tokens
Parsing

- The parser converts text into expressions

Lexical Analysis

Tokens

Syntactic Analysis

Expressions

'(+ 1'
'   (- 23)' 
'   (* 4 5.6))'

[', '+', 1
'(', '-', 23, ')
'(', '*', 4, 5.6, '), ')]

- Iterative process
- Checks number of parentheses
- Checks for malformed tokens
- Determines types of tokens
The parser converts text into expressions.

- Iterative process
- Checks number of parentheses
- Checks for malformed tokens
- Determines types of tokens
Parsing

- The parser converts text into expressions

Lexical Analysis

Expressions

Text

Tokens

• Iterative process
• Checks number of parentheses
• Checks for malformed tokens
• Determines types of tokens

'(+ 1'
'  (- 23)'
'  (* 4 5.6))'

[', '+', 1
'(', '-', 23, ')
'(', '*', 4, 5.6, ')', ')']

Pair('+', Pair(1, ...))

printed as

(+ 1 (- 23) (* 4 5.6))
The parser converts text into expressions.

- Iterative process
- Checks number of parentheses
- Checks for malformed tokens
- Determines types of tokens

Text

 Lexical Analysis

'(+ 1'  '(- 23)'  '(* 4 5.6)')

Tokens

 ['(', '+', 1 '(', '-', 23, ')'] ['(', '*', 4, 5.6, ')', ')']

Expressions

Pair('+', Pair(1, ...))

( + 1 (- 23) (* 4 5.6))

• Tree-recursive process
Parsing

• The parser converts text into expressions

Text

Lexical Analysis

Tokens

Syntactic Analysis

Expressions

'(+ 1
'  (- 23)'
'  (* 4 5.6))'

[('(', '+', 1
  ('-', '23', ')')
  ('*', '4', '5.6', ')', ')')]

Pair('+', Pair(1, ...))

printed as

(+ 1 (- 23) (* 4 5.6))

• Iterative process
• Checks number of parentheses
• Checks for malformed tokens
• Determines types of tokens

• Tree-recursive process
• Processes tokens one by one
Parsing

• The parser converts text into expressions

Text

Lexical Analysis

Tokens

Syntactic Analysis

Expressions

'(+ 1'
'   (- 23)' 
'   (* 4 5.6))'

[', '+', 1
'   ('', '- ', 23, ')
'   ('', '* ', 4, 5.6, '), ')', ')]

Pair('+', Pair(1, ...))

(printed as)

(+ 1 (- 23) (* 4 5.6))

• Iterative process
• Checks number of parentheses
• Checks for malformed tokens
• Determines types of tokens

• Tree-recursive process
• Processes tokens one by one
• Checks parenthesis structure
 Parsing

- The parser converts text into expressions

\[ \text{Pair('+', Pair(1, \ldots))} \]

**Lexical Analysis**

- Iterative process
- Checks number of parentheses
- Checks for malformed tokens
- Determines types of tokens

**Syntactic Analysis**

- Tree-recursive process
- Processes tokens one by one
- Checks parenthesis structure
- Returns expression as a Pair

\[ ('+', '+', 1 \text{Pair('-', 23, ')')*4, 5.6)'\]
Lexical Analysis
Lexical Analysis

- Tokenization takes in a string and converts it into a list of tokens by splitting on whitespace
Lexical Analysis

- Tokenization takes in a string and converts it into a list of tokens by splitting on whitespace
- This step also removes excess whitespace
Lexical Analysis

- Tokenization takes in a string and converts it into a list of tokens by splitting on whitespace
  - This step also removes excess whitespace

- An error is raised if the number of open and closed parentheses are unequal
Lexical Analysis

- Tokenization takes in a string and converts it into a list of tokens by splitting on whitespace
  - This step also removes excess whitespace

- An error is raised if the number of open and closed parentheses are unequal

- Each token is checked iteratively to ensure it is valid
Lexical Analysis

- Tokenization takes in a string and converts it into a list of tokens by splitting on whitespace
  - This step also removes excess whitespace

- An error is raised if the number of open and closed parentheses are unequal

- Each token is checked iteratively to ensure it is valid
  - For Calculator, each token must be a parenthesis, an operator, or a number
Lexical Analysis

• Tokenization takes in a string and converts it into a list of tokens by splitting on whitespace
  • This step also removes excess whitespace

• An error is raised if the number of open and closed parentheses are unequal

• Each token is checked iteratively to ensure it is valid
  • For Calculator, each token must be a parenthesis, an operator, or a number
  • Otherwise, an error is raised
Lexical Analysis

- Tokenization takes in a string and converts it into a list of tokens by splitting on whitespace
  - This step also removes excess whitespace

- An error is raised if the number of open and closed parentheses are unequal

- Each token is checked iteratively to ensure it is valid
  - For Calculator, each token must be a parenthesis, an operator, or a number
  - Otherwise, an error is raised
Syntactic Analysis
Syntactic Analysis

- Syntactic analysis uses a *read function* to identify the hierarchical structure of an expression
Syntactic Analysis

• Syntactic analysis uses a *read function* to identify the hierarchical structure of an expression

• Each call to the read function consumes the input tokens for exactly one expression, and returns the expression
Syntactic Analysis (demo)

- Syntactic analysis uses a *read function* to identify the hierarchical structure of an expression.
- Each call to the read function consumes the input tokens for exactly one expression, and returns the expression.
Syntactic Analysis

- Syntactic analysis uses a read function to identify the hierarchical structure of an expression.
- Each call to the read function consumes the input tokens for exactly one expression, and returns the expression.

```python
def read_exp(tokens):
    """Returns the first calculator expression.""
    ...

def read_tail(tokens):
    """Reads up to the first mismatched close parenthesis.""
    ...
```
Syntactic Analysis

- Syntactic analysis uses a read function to identify the hierarchical structure of an expression.
- Each call to the read function consumes the input tokens for exactly one expression, and returns the expression.

```python
def read_exp(tokens):
    """Returns the first calculator expression."""
    ...

def read_tail(tokens):
    """Reads up to the first mismatched close parenthesis."""
    ...

['(' ', '+', 1, '(', '−', 23, ')', '(', '*', 4, 5.6, ')', ')']
Syntactic Analysis

- Syntactic analysis uses a read function to identify the hierarchical structure of an expression.
- Each call to the read function consumes the input tokens for exactly one expression, and returns the expression.

```python
def read_exp(tokens):
    """Returns the first calculator expression."""
    ...

def read_tail(tokens):
    """Reads up to the first mismatched close parenthesis."""
    ...
```

```
[('(', '+', 1, '(', '-', 23, ')', '(', '*', 4, 5.6, ')', ')]
```

Resulting expression:
Syntactic Analysis

• Syntactic analysis uses a *read function to* identify the hierarchical structure of an expression

• Each call to the read function consumes the input tokens for exactly one expression, and returns the expression

```python
def read_exp(tokens):
    """Returns the first calculator expression.""
    ...

def read_tail(tokens):
    """Reads up to the first mismatched close parenthesis.""
    ...
```

Resulting expression: (
Syntactic Analysis

• Syntactic analysis uses a *read function* to identify the hierarchical structure of an expression

• Each call to the read function consumes the input tokens for exactly one expression, and returns the expression

```python
def read_exp(tokens):
    """Returns the first calculator expression."""
    ...

def read_tail(tokens):
    """Reads up to the first mismatched close parenthesis."""
    ...
```

```python
token_list = ['+', 1, '(', '-', 23, ')', '(' , '*', 4, 5.6, ')', ']
Resulting expression: (
Syntactic Analysis

- Syntactic analysis uses a *read function* to identify the hierarchical structure of an expression.
- Each call to the read function consumes the input tokens for exactly one expression, and returns the expression.

```python
def read_exp(tokens):
    """Returns the first calculator expression.""
    ...

def read_tail(tokens):
    """Reads up to the first mismatched close parenthesis.""
    ...
```

Resulting expression:

```
[1, '(', '-', 23, ')', '(', '*', 4, 5.6, ')', ', ')']
```

Resulting expression: (+
Syntactic Analysis

- Syntactic analysis uses a read function to identify the hierarchical structure of an expression.
- Each call to the read function consumes the input tokens for exactly one expression, and returns the expression.

```python
def read_exp(tokens):
    """Returns the first calculator expression.""
    ...

def read_tail(tokens):
    """Reads up to the first mismatched close parenthesis.""
    ...
```

```
[1, '(', '-', 23, '), (' ', '*', 4, 5.6, '), )']
```

Resulting expression: (+
Syntactic Analysis

• Syntactic analysis uses a *read function to* identify the hierarchical structure of an expression

• Each call to the read function consumes the input tokens for exactly one expression, and returns the expression

```python
def read_exp(tokens):
    """Returns the first calculator expression."""
    ...

def read_tail(tokens):
    """Reads up to the first mismatched close parenthesis."""
    ...
```

Resulting expression: (+ 1
Syntactic Analysis

- Syntactic analysis uses a **read function to** identify the hierarchical structure of an expression.
- Each call to the read function consumes the input tokens for exactly one expression, and returns the expression.

```python
def read_exp(tokens):
    """Returns the first calculator expression.""
    ...

def read_tail(tokens):
    """Reads up to the first mismatched close parenthesis.""
    ...
```

Input tokens:

```
[('(', '-', 23, ')', '(', '*', 4, 5.6, ')', ')]
```

Resulting expression: (+ 1)
Syntactic Analysis

- Syntactic analysis uses a *read function* to identify the hierarchical structure of an expression.
- Each call to the read function consumes the input tokens for exactly one expression, and returns the expression.

```python
def read_exp(tokens):
    """Returns the first calculator expression."""
    ...

def read_tail(tokens):
    """Reads up to the first mismatched close parenthesis."""
    ...

['(', '*', 4, 5.6, ')', ')']

Resulting expression: (+ 1 (- 23))
Syntactic Analysis

- Syntactic analysis uses a *read function to* identify the hierarchical structure of an expression.
- Each call to the read function consumes the input tokens for exactly one expression, and returns the expression.

```python
def read_exp(tokens):
    """Returns the first calculator expression.""
    ...

def read_tail(tokens):
    """Reads up to the first mismatched close parenthesis.""
    ...
```

Resulting expression:

```python
[(', ', '*', 4, 5.6, '), ')'
```
Syntactic Analysis

• Syntactic analysis uses a read function to identify the hierarchical structure of an expression

• Each call to the read function consumes the input tokens for exactly one expression, and returns the expression

```python
def read_exp(tokens):
    """Returns the first calculator expression.""
    ...

def read_tail(tokens):
    """Reads up to the first mismatched close parenthesis.""
    ...
```

Resulting expression: (+ 1 (- 23) (* 4 5.6))
Syntactic Analysis

- Syntactic analysis uses a read function to identify the hierarchical structure of an expression.
- Each call to the read function consumes the input tokens for exactly one expression, and returns the expression.

```python
def read_exp(tokens):
    """Returns the first calculator expression.""
    ...

def read_tail(tokens):
    """Reads up to the first mismatched close parenthesis.""
    ...
```

Resulting expression: \((+ 1 (- 23) (* 4 5.6))\)
Evaluation

From expressions to values
Evaluation

• Evaluation is performed by an evaluate function, which takes in an expression (the output of our parser) and computes and returns the value of the expression
Evaluation

- Evaluation is performed by an evaluate function, which takes in an expression (the output of our parser) and computes and returns the value of the expression
  - In Calculator, the value is always an operator or a number
Evaluation

- Evaluation is performed by an evaluate function, which takes in an expression (the output of our parser) and computes and returns the value of the expression.
  - In Calculator, the value is always an operator or a number.

- If the expression is primitive, we can return the value of the expression directly.
Evaluation

Evaluation is performed by an evaluate function, which takes in an expression (the output of our parser) and computes and returns the value of the expression.

- In Calculator, the value is always an operator or a number.

- If the expression is primitive, we can return the value of the expression directly.

- Otherwise, we have a call expression, and we follow the rules for evaluating call expressions:
Evaluation

• Evaluation is performed by an evaluate function, which takes in an expression (the output of our parser) and computes and returns the value of the expression
  • In Calculator, the value is always an operator or a number

• If the expression is primitive, we can return the value of the expression directly

• Otherwise, we have a call expression, and we follow the rules for evaluating call expressions:
  1. Evaluate the operator to get a function
Evaluation

- Evaluation is performed by an evaluate function, which takes in an expression (the output of our parser) and computes and returns the value of the expression
  - In Calculator, the value is always an operator or a number

- If the expression is primitive, we can return the value of the expression directly

- Otherwise, we have a call expression, and we follow the rules for evaluating call expressions:
  1. Evaluate the operator to get a function
  2. Evaluate the operands to get its values
Evaluation

- Evaluation is performed by an *evaluate function*, which takes in an expression (the output of our parser) and computes and returns the value of the expression
  - In Calculator, the value is always an operator or a number

- If the expression is primitive, we can return the value of the expression directly

- Otherwise, we have a call expression, and we follow the rules for evaluating call expressions:
  1. *Evaluate* the operator to get a function
  2. *Evaluate* the operands to get its values
  3. *Apply* the function to the values of the operands to get the final value
Evaluation

- Evaluation is performed by an evaluate function, which takes in an expression (the output of our parser) and computes and returns the value of the expression
  - In Calculator, the value is always an operator or a number

- If the expression is primitive, we can return the value of the expression directly

- Otherwise, we have a call expression, and we follow the rules for evaluating call expressions:
  1. Evaluate the operator to get a function
  2. Evaluate the operands to get its values
  3. Apply the function to the values of the operands to get the final value
  - This hopefully looks very familiar!
Evaluation

• Evaluation is performed by an evaluate function, which takes in an expression (the output of our parser) and computes and returns the value of the expression
  • In Calculator, the value is always an operator or a number

• If the expression is primitive, we can return the value of the expression directly

• Otherwise, we have a call expression, and we follow the rules for evaluating call expressions:
  1. Evaluate the operator to get a function
  2. Evaluate the operands to get its values
  3. Apply the function to the values of the operands to get the final value

• This hopefully looks very familiar!
The Evaluate and Apply Functions
def calc_eval(exp):
    if isinstance(exp, Pair):
        return calc_apply(calc_eval(exp.first),
                           list(exp.second.map(calc_eval)))
    elif exp in OPERATORS:
        return OPERATORS[exp]
    else:
        return exp

def calc_apply(op, args):
    return op(*args)
The Evaluate and Apply Functions

def calc_eval(exp):
    if isinstance(exp, Pair):
        return calc_apply(calc_eval(exp.first),
                           list(exp.second.map(calc_eval)))
    elif exp in OPERATORS:
        return OPERATORS[exp]
    else:
        return exp
def calc_apply(op, args):
    return op(*args)

• Why define calc_apply? It’s not really necessary, since the Calculator language is so simple
The Evaluate and Apply Functions

```python
def calc_eval(exp):
    if isinstance(exp, Pair):
        return calc_apply(calc_eval(exp.first),
                           list(exp.second.map(calc_eval)))
    elif exp in OPERATORS:
        return OPERATORS[exp]
    else:
        return exp

def calc_apply(op, args):
    return op(*args)
```

- Why define `calc_apply`? It’s not really necessary, since the Calculator language is so simple
- For real languages, applying functions is more complex
The Evaluate and Apply Functions

```python
def calc_eval(exp):
    if isinstance(exp, Pair):
        return calc_apply(calc_eval(exp.first),
                           list(exp.second.map(calc_eval)))
    elif exp in OPERATORS:
        return OPERATORS[exp]
    else:
        return exp

def calc_apply(op, args):
    return op(*args)
```

• Why define `calc_apply`? It’s not really necessary, since
  the Calculator language is so simple

• For real languages, applying functions is more complex

• With user-defined functions, the apply function has to
  call the evaluate function! This mutual recursion is
  called the `eval-apply loop`
Putting it all together

A Calculator interactive interpreter!
The Read-Eval-Print Loop
The Read-Eval-Print Loop

- Interactive interpreters all follow the same interface:
The Read-Eval-Print Loop

• Interactive interpreters all follow the same interface:
  1. Print a prompt
The Read-Eval-Print Loop

- Interactive interpreters all follow the same interface:
  1. Print a prompt
  2. Read text input from the user
The Read–Eval–Print Loop

- Interactive interpreters all follow the same interface:
  1. Print a prompt
  2. Read text input from the user
  3. Parse the input into an expression
The Read-Eval-Print Loop

Interactive interpreters all follow the same interface:

1. Print a prompt
2. Read text input from the user
3. Parse the input into an expression
4. Evaluate the expression into a value
The Read–Eval–Print Loop

- Interactive interpreters all follow the same interface:
  1. Print a prompt
  2. Read text input from the user
  3. Parse the input into an expression
  4. Evaluate the expression into a value
  5. Report any errors, if they occur, otherwise
The Read-Eval-Print Loop

- Interactive interpreters all follow the same interface:
  1. Print a prompt
  2. Read text input from the user
  3. Parse the input into an expression
  4. Evaluate the expression into a value
  5. Report any errors, if they occur, otherwise
  6. Print the value and return to step 1
The Read-Eval-Print Loop

- Interactive interpreters all follow the same interface:
  1. Print a prompt
  2. Read text input from the user
  3. Parse the input into an expression
  4. Evaluate the expression into a value
  5. Report any errors, if they occur, otherwise
  6. Print the value and return to step 1

- This is known as the read-eval-print loop (REPL)
The Read–Eval–Print Loop

- Interactive interpreters all follow the same interface:
  1. Print a prompt
  2. Read text input from the user
  3. Parse the input into an expression
  4. Evaluate the expression into a value
  5. Report any errors, if they occur, otherwise
  6. Print the value and return to step 1

- This is known as the read–eval–print loop (REPL)
Handling Exceptions
Handling Exceptions

- Various exceptions may be raised throughout the REPL:
Handling Exceptions

• Various exceptions may be raised throughout the REPL:
  • **Lexical analysis:** The token `2.3.4` raises `SyntaxError`
Handling Exceptions

• Various exceptions may be raised throughout the REPL:
  • **Lexical analysis:** The token 2.3.4 raises `SyntaxError`
  • **Syntactic analysis:** A misplaced `)` raises `SyntaxError`
Handling Exceptions

• Various exceptions may be raised throughout the REPL:
  • **Lexical analysis:** The token 2.3.4 raises `SyntaxError`
  • **Syntactic analysis:** A misplaced ) raises `SyntaxError`
  • **Evaluation:** No arguments to - raises `TypeError`
Handling Exceptions

• Various exceptions may be raised throughout the REPL:
  • **Lexical analysis:** The token 2.3.4 raises `SyntaxError`
  • **Syntactic analysis:** A misplaced `)` raises `SyntaxError`
  • **Evaluation:** No arguments to `-` raises `TypeError`

• An interactive interpreter prints information about each error that occurs
Handling Exceptions

- Various exceptions may be raised throughout the REPL:
  - **Lexical analysis:** The token 2.3.4 raises `SyntaxError`
  - **Syntactic analysis:** A misplaced `)` raises `SyntaxError`
  - **Evaluation:** No arguments to `-` raises `TypeError`

- An interactive interpreter prints information about each error that occurs

- A well-designed interactive interpreter should not halt completely on an error, so that the user has an opportunity to try again in the current environment
Handling Exceptions

• Various exceptions may be raised throughout the REPL:
  • **Lexical analysis**: The token 2.3.4 raises SyntaxError
  • **Syntactic analysis**: A misplaced ) raises SyntaxError
  • **Evaluation**: No arguments to - raises TypeError

• An interactive interpreter prints information about each error that occurs

• A well-designed interactive interpreter should not halt completely on an error, so that the user has an opportunity to try again in the current environment
Summary
Summary

• We built an interpreter today!
Summary

• We built an interpreter today!
  • It was for a very simple language, but the same ideas and principles will allow us to build an interpreter for Scheme, a much more complicated language
Summary

- We built an interpreter today!
  - It was for a very simple language, but the same ideas and principles will allow us to build an interpreter for Scheme, a much more complicated language
  - More complicated examples are coming soon
Summary

• We built an interpreter today!
  • It was for a very simple language, but the same ideas and principles will allow us to build an interpreter for Scheme, a much more complicated language
  • More complicated examples are coming soon

• Interpreters are separated into a parser and an evaluator
Summary

• We built an interpreter today!
  • It was for a very simple language, but the same ideas and principles will allow us to build an interpreter for Scheme, a much more complicated language
  • More complicated examples are coming soon

• Interpreters are separated into a parser and an evaluator
  • The parser takes in text input and outputs the corresponding expressions, using tokens as a midpoint
Summary

• We built an interpreter today!
  • It was for a very simple language, but the same ideas and principles will allow us to build an interpreter for Scheme, a much more complicated language
  • More complicated examples are coming soon

• Interpreters are separated into a parser and an evaluator
  • The parser takes in text input and outputs the corresponding expressions, using tokens as a midpoint
  • The evaluator takes in an expression and outputs the corresponding value
Summary

• We built an interpreter today!
  • It was for a very simple language, but the same ideas and principles will allow us to build an interpreter for Scheme, a much more complicated language
  • More complicated examples are coming soon

• Interpreters are separated into a parser and an evaluator
  • The parser takes in text input and outputs the corresponding expressions, using tokens as a midpoint
  • The evaluator takes in an expression and outputs the corresponding value
  • The read–eval–print loop completes our interpreter