61A Lecture 26
Announcements
Programming Languages
Programming Languages
Programming Languages

A computer typically executes programs written in many different programming languages
Programming Languages

A computer typically executes programs written in many different programming languages.

**Machine languages**: statements are interpreted by the hardware itself.
Programming Languages

A computer typically executes programs written in many different programming languages

**Machine languages**: statements are interpreted by the hardware itself

- A fixed set of instructions invoke operations implemented by the circuitry of the central processing unit (CPU)
Programming Languages

A computer typically executes programs written in many different programming languages.

**Machine languages:** statements are interpreted by the hardware itself

- A fixed set of instructions invoke operations implemented by the circuitry of the central processing unit (CPU)
- Operations refer to specific hardware memory addresses; no abstraction mechanisms
Programming Languages

A computer typically executes programs written in many different programming languages.

**Machine languages**: statements are interpreted by the hardware itself
- A fixed set of instructions invoke operations implemented by the circuitry of the central processing unit (CPU)
- Operations refer to specific hardware memory addresses; no abstraction mechanisms

**High-level languages**: statements & expressions are interpreted by another program or compiled (translated) into another language
Programming Languages

A computer typically executes programs written in many different programming languages

**Machine languages:** statements are interpreted by the hardware itself

- A fixed set of instructions invoke operations implemented by the circuitry of the central processing unit (CPU)
- Operations refer to specific hardware memory addresses; no abstraction mechanisms

**High-level languages:** statements & expressions are interpreted by another program or compiled (translated) into another language

- Provide means of abstraction such as naming, function definition, and objects
Programming Languages

A computer typically executes programs written in many different programming languages.

**Machine languages**: statements are interpreted by the hardware itself
- A fixed set of instructions invoke operations implemented by the circuitry of the central processing unit (CPU)
- Operations refer to specific hardware memory addresses; no abstraction mechanisms

**High-level languages**: statements & expressions are interpreted by another program or compiled (translated) into another language
- Provide means of abstraction such as naming, function definition, and objects
- Abstract away system details to be independent of hardware and operating system
Programming Languages

A computer typically executes programs written in many different programming languages

**Machine languages:** statements are interpreted by the hardware itself

- A fixed set of instructions invoke operations implemented by the circuitry of the central processing unit (CPU)
- Operations refer to specific hardware memory addresses; no abstraction mechanisms

**High-level languages:** statements & expressions are interpreted by another program or compiled (translated) into another language

- Provide means of abstraction such as naming, function definition, and objects
- Abstract away system details to be independent of hardware and operating system

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

Python 3
Programming Languages

A computer typically executes programs written in many different programming languages

**Machine languages**: statements are interpreted by the hardware itself

- A fixed set of instructions invoke operations implemented by the circuitry of the central processing unit (CPU)
- Operations refer to specific hardware memory addresses; no abstraction mechanisms

**High-level languages**: statements & expressions are interpreted by another program or compiled (translated) into another language

- Provide means of abstraction such as naming, function definition, and objects
- Abstract away system details to be independent of hardware and operating system

```python
Python 3

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

```python 3 Byte Code

```

```

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

A computer typically executes programs written in many different programming languages.

**Machine languages**: statements are interpreted by the hardware itself
- A fixed set of instructions invoke operations implemented by the circuitry of the central processing unit (CPU)
- Operations refer to specific hardware memory addresses; no abstraction mechanisms

**High-level languages**: statements & expressions are interpreted by another program or compiled (translated) into another language
- Provide means of abstraction such as naming, function definition, and objects
- Abstract away system details to be independent of hardware and operating system

```
Python 3

```def square(x):
    return x * x
```from dis import dis
dis(square)
```
Python 3 Byte Code

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

A powerful form of abstraction is to define a new language that is tailored to a particular type of application or problem domain.
Metalinguistic Abstraction

A powerful form of abstraction is to define a new language that is tailored to a particular type of application or problem domain

**Type of application:** Erlang was designed for concurrent programs. It has built-in elements for expressing concurrent communication. It is used, for example, to implement chat servers with many simultaneous connections
Metalinguistic Abstraction

A powerful form of abstraction is to define a new language that is tailored to a particular type of application or problem domain.

**Type of application:** Erlang was designed for concurrent programs. It has built-in elements for expressing concurrent communication. It is used, for example, to implement chat servers with many simultaneous connections.

**Problem domain:** The MediaWiki mark-up language was designed for generating static web pages. It has built-in elements for text formatting and cross-page linking. It is used, for example, to create Wikipedia pages.
Metalinguistic Abstraction

A powerful form of abstraction is to define a new language that is tailored to a particular type of application or problem domain.

**Type of application:** Erlang was designed for concurrent programs. It has built-in elements for expressing concurrent communication. It is used, for example, to implement chat servers with many simultaneous connections.

**Problem domain:** The MediaWiki mark-up language was designed for generating static web pages. It has built-in elements for text formatting and cross-page linking. It is used, for example, to create Wikipedia pages.

A programming language has:
Metalinguistic Abstraction

A powerful form of abstraction is to define a new language that is tailored to a particular type of application or problem domain.

**Type of application:** Erlang was designed for concurrent programs. It has built-in elements for expressing concurrent communication. It is used, for example, to implement chat servers with many simultaneous connections.

**Problem domain:** The MediaWiki mark-up language was designed for generating static web pages. It has built-in elements for text formatting and cross-page linking. It is used, for example, to create Wikipedia pages.

A programming language has:

- **Syntax:** The legal statements and expressions in the language.
A powerful form of abstraction is to define a new language that is tailored to a particular type of application or problem domain.

**Type of application:** Erlang was designed for concurrent programs. It has built-in elements for expressing concurrent communication. It is used, for example, to implement chat servers with many simultaneous connections.

**Problem domain:** The MediaWiki mark-up language was designed for generating static web pages. It has built-in elements for text formatting and cross-page linking. It is used, for example, to create Wikipedia pages.

A programming language has:

- **Syntax:** The legal statements and expressions in the language
- **Semantics:** The execution/evaluation rule for those statements and expressions
Metalinguistic Abstraction

A powerful form of abstraction is to define a new language that is tailored to a particular type of application or problem domain

**Type of application:** Erlang was designed for concurrent programs. It has built-in elements for expressing concurrent communication. It is used, for example, to implement chat servers with many simultaneous connections

**Problem domain:** The MediaWiki mark-up language was designed for generating static web pages. It has built-in elements for text formatting and cross-page linking. It is used, for example, to create Wikipedia pages

A programming language has:

- **Syntax:** The legal statements and expressions in the language
- **Semantics:** The execution/evaluation rule for those statements and expressions

To create a new programming language, you either need a:
Metalinguistic Abstraction

A powerful form of abstraction is to define a new language that is tailored to a particular type of application or problem domain

**Type of application:** Erlang was designed for concurrent programs. It has built-in elements for expressing concurrent communication. It is used, for example, to implement chat servers with many simultaneous connections

**Problem domain:** The MediaWiki mark-up language was designed for generating static web pages. It has built-in elements for text formatting and cross-page linking. It is used, for example, to create Wikipedia pages

A programming language has:

- **Syntax:** The legal statements and expressions in the language
- **Semantics:** The execution/evaluation rule for those statements and expressions

To create a new programming language, you either need a:

- **Specification:** A document describe the precise syntax and semantics of the language
Metalinguistic Abstraction

A powerful form of abstraction is to define a new language that is tailored to a particular type of application or problem domain

**Type of application:** Erlang was designed for concurrent programs. It has built-in elements for expressing concurrent communication. It is used, for example, to implement chat servers with many simultaneous connections.

**Problem domain:** The MediaWiki mark-up language was designed for generating static web pages. It has built-in elements for text formatting and cross-page linking. It is used, for example, to create Wikipedia pages.

A programming language has:

- **Syntax:** The legal statements and expressions in the language
- **Semantics:** The execution/evaluation rule for those statements and expressions

To create a new programming language, you either need a:

- **Specification:** A document describe the precise syntax and semantics of the language
- **Canonical Implementation:** An interpreter or compiler for the language
Parsing
Reading Scheme Lists
Reading Scheme Lists

A Scheme list is written as elements in parentheses:
Reading Scheme Lists

A Scheme list is written as elements in parentheses:

$$(\text{<element_0> \ <element_1> \ldots \ <element_n>})$$
Reading Scheme Lists

A Scheme list is written as elements in parentheses:

(`<element_0> <element_1> ... <element_n>`)  

Each `<element>` can be a combination or primitive
Reading Scheme Lists

A Scheme list is written as elements in parentheses:

\((<element_0> \ <element_1> \ ... \ <element_n>)\)

Each \(<element>\) can be a combination or primitive

\((+ (\times 3 (\times (* 2 4) (+ 3 5))) (+ (- 10 7) 6))\)
Reading Scheme Lists

A Scheme list is written as elements in parentheses:

\[(\text{element}_0\ \text{element}_1\ \ldots\ \text{element}_n)\]

Each element can be a combination or primitive

\[(+\ (+\ (\ast\ 3\ (+\ (\ast\ 2\ 4)\ (+\ 3\ 5)))\ (+\ (-10\ 7)\ 6))\]

The task of parsing a language involves coercing a string representation of an expression to the expression itself
Reading Scheme Lists

A Scheme list is written as elements in parentheses:

\((<\text{element}_0> <\text{element}_1> \ldots <\text{element}_n>)\)

Each \(<\text{element}>\) can be a combination or primitive

\((+ (* 3 (+ (* 2 4) (+ 3 5))) (+ (- 10 7) 6))\)

The task of parsing a language involves coercing a string representation of an expression to the expression itself

(Demo)

http://composingprograms.com/examples/scalc/scheme_reader.py.html
Reading Scheme Lists

A Scheme list is written as elements in parentheses:

```
(<element_0> <element_1> ... <element_n>)
```

Each <element> can be a combination or primitive

```
(+ (* 3 (+ (* 2 4) (+ 3 5))) (+ (- 10 7) 6))
```

The task of parsing a language involves coercing a string representation of an expression to the expression itself

(Demo)

http://composingprograms.com/examples/scalc/scheme_reader.py.html
Reading Scheme Lists

A Scheme list is written as elements in parentheses:

\[(\text{element}_0 \text{ element}_1 \ldots \text{element}_n)\]

Each <element> can be a combination or primitive

\[ (+ (* 3 (+ (* 2 4) (+ 3 5))) (+ (- 10 7) 6)) \]

The task of parsing a language involves coercing a string representation of an expression to the expression itself

(Demo)

http://composingprograms.com/examples/scalc/scheme_reader.py.html
Reading Scheme Lists

A Scheme list is written as elements in parentheses:

\[(element_0) (element_1) \ldots (element_n)\]

Each <element> can be a combination or primitive

\[(+ (* 3 (+ (* 2 4) (+ 3 5))) (+ (- 10 7) 6))\]

The task of parsing a language involves coercing a string representation of an expression to the expression itself

(Demo)

http://composingprograms.com/examples/scalc/scheme_reader.py.html
A Parser takes text and returns an expression.
A Parser takes text and returns an expression
A Parser takes text and returns an expression.
A Parser takes text and returns an expression
Parsing

A Parser takes text and returns an expression

Text → Lexical analysis → Tokens → Syntactic analysis → Expression
A Parser takes text and returns an expression

'(+ 1'  
'   (- 23)'  
'   (* 4 5.6))'
A Parser takes text and returns an expression

Text → Lexical analysis → Tokens → Syntactic analysis → Expression

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

A Parser takes text and returns an expression

Text → Lexical analysis → Tokens → Syntactic analysis → Expression

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

'(+, '+', 1
'(', '+', 1
A Parser takes text and returns an expression.
A Parser takes text and returns an expression

Text

Lexical analysis

Tokens

Syntactic analysis

Expression

'(+ 1'
'(− 23)'
'(* 4 5.6))'

'(', '+', 1
'(−', '+', 23, ')
'(')
A Parser takes text and returns an expression.
A Parser takes text and returns an expression.
A Parser takes text and returns an expression

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

Lexical analysis: '(' '+', 1  
                  '(' '-', 23, ')'  
                  '(' '*' 4, 5.6, ')')'

Tokens: '(' '+', 1  
        '(' '-', 23, ')'  
        '(' '*' 4, 5.6, ')')'

Syntactic analysis: '(' '+', 1  
                      '(' '-', 23, ')'  
                      '(' '*' 4, 5.6, ')')'
A Parser takes text and returns an expression

- Parsing
  - Iterative process
A Parser takes text and returns an expression

- Iterative process
- Checks for malformed tokens
A Parser takes text and returns an expression

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

A Parser takes text and returns an expression

- **Lexical analysis**
  - Iterative process
  - Checks for malformed tokens
  - Determines types of tokens
  - Processes one line at a time

- **Tokens**

- **Syntactic analysis**

- **Expression**

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

Expression: 
'(', '+', 1
'(', '-', 23, ')'
'(', '*', 4, 5.6, ')', ')'

Diagram:

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

- Tokens: '(', '+', 1
  '(', '-', 23, ')'
  '(', '*', 4, 5.6, ')', ')

- Syntactic analysis

- Expression
A Parser takes text and returns an expression

- **Lexical analysis**:
  - Iterative process
  - Checks for malformed tokens
  - Determines types of tokens
  - Processes one line at a time

- **Syntactic analysis**

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

Tokens: 
```
(('(', '+', 1
  ('(', '-', 23, ')
  ('(', '*', 4, 5.6, ')
  ')
)')
```
 Parsing

A Parser takes text and returns an expression

- Lexical analysis
  - Iterative process
  - Checks for malformed tokens
  - Determines types of tokens
  - Processes one line at a time

- Syntactic analysis

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

- Tokens: '
  (', '+', 1
  (', '-', 23, ')
  (', '*', 4, 5.6, ')', ')

- Expression: Pair('+', Pair(1, ...))
A Parser takes text and returns an expression

- Iterative process
- Checks for malformed tokens
- Determines types of tokens
- Processes one line at a time
Parsing

A Parser takes text and returns an expression

<table>
<thead>
<tr>
<th>Text</th>
<th>Lexical analysis</th>
<th>Tokens</th>
<th>Syntactic analysis</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>'(+ 1'</td>
<td></td>
<td>'(' '+', 1</td>
<td></td>
<td>Pair('+', Pair(1, ...))</td>
</tr>
<tr>
<td>'(' '-', 23, ')'</td>
<td></td>
<td>'(' '*', 4, 5.6, ')'</td>
<td></td>
<td>(+ 1 (- 23) (* 4 5.6))</td>
</tr>
<tr>
<td>'(' '*', 4, 5.6, ')'</td>
<td></td>
<td></td>
<td></td>
<td>printed as</td>
</tr>
</tbody>
</table>

• Iterative process
• Checks for malformed tokens
• Determines types of tokens
• Processes one line at a time

• Tree-recursive process
A Parser takes text and returns an expression

- Iterative process
- Checks for malformed tokens
- Determines types of tokens
- Processes one line at a time

- Tree-recursive process
- Balances parentheses
A Parser takes text and returns an expression

<table>
<thead>
<tr>
<th>Text</th>
<th>Lexical analysis</th>
<th>Tokens</th>
<th>Syntactic analysis</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>'(+ 1'</td>
<td>'('</td>
<td>'+'</td>
<td>1</td>
<td>'Pair('+' , Pair(1, ...))</td>
</tr>
<tr>
<td>'(- 23)'</td>
<td>'('</td>
<td>'-'</td>
<td>23</td>
<td>')'</td>
</tr>
<tr>
<td>'(* 4 5.6)')'</td>
<td>'('</td>
<td>'*'</td>
<td>4, 5.6</td>
<td>')'</td>
</tr>
<tr>
<td>'('</td>
<td>'('</td>
<td>'(', '&lt;double_process&gt;', 1</td>
<td>)'</td>
<td></td>
</tr>
<tr>
<td>'('</td>
<td>'(', 'double_process&gt;', 23</td>
<td>', ')'</td>
<td></td>
<td></td>
</tr>
<tr>
<td>'('</td>
<td>'(', 'double_process&gt;', 4, 5.6</td>
<td>', ')'</td>
<td></td>
<td></td>
</tr>
<tr>
<td>'('</td>
<td>'(', 'double_process&gt;', 1</td>
<td>')'</td>
<td></td>
<td></td>
</tr>
<tr>
<td>'('</td>
<td>'(', 'double_process&gt;', 23</td>
<td>', ')'</td>
<td></td>
<td></td>
</tr>
<tr>
<td>'('</td>
<td>'(', 'double_process&gt;', 4, 5.6</td>
<td>', ')'</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Iterative process
- Checks for malformed tokens
- Determines types of tokens
- Processes one line at a time

- Tree-recursive process
- Balances parentheses
- Returns tree structure
Parsing

A Parser takes text and returns an expression

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

<table>
<thead>
<tr>
<th>Text</th>
<th>Lexical analysis</th>
<th>Tokens</th>
<th>Syntactic analysis</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>'(+ 1'</td>
<td></td>
<td>'(' '', '+', 1</td>
<td></td>
<td>Pair('+', Pair(1, ...))</td>
</tr>
<tr>
<td>'(- 23)'</td>
<td></td>
<td>'(' '', '-', 23, ')'</td>
<td></td>
<td>printed as</td>
</tr>
<tr>
<td>'(* 4 5.6)')</td>
<td></td>
<td>'(' '', '*', 4, 5.6, ')'</td>
<td></td>
<td>(+ 1 (- 23) (* 4 5.6))</td>
</tr>
</tbody>
</table>

- Iterative process
- Checks for malformed tokens
- Determines types of tokens
- Processes one line at a time

- Tree-recursive process
- Balances parentheses
- Returns tree structure
- Processes multiple lines
Syntactic Analysis
Syntactic Analysis

Syntactic analysis identifies the hierarchical structure of an expression, which may be nested.
Syntactic Analysis

Syntactic analysis identifies the hierarchical structure of an expression, which may be nested.

Each call to scheme_read consumes the input tokens for exactly one expression.
Syntactic Analysis

Syntactic analysis identifies the hierarchical structure of an expression, which may be nested.

Each call to scheme_read consumes the input tokens for exactly one expression.

**Base case:** symbols and numbers
**Syntactic Analysis**

Syntactic analysis identifies the hierarchical structure of an expression, which may be nested.

Each call to `scheme_read` consumes the input tokens for exactly one expression.

- **Base case:** symbols and numbers
- **Recursive call:** `scheme_read` sub-expressions and combine them.
Syntactic Analysis

Syntactic analysis identifies the hierarchical structure of an expression, which may be nested.

Each call to scheme_read consumes the input tokens for exactly one expression.

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

**Base case:** symbols and numbers

**Recursive call:** scheme_read sub-expressions and combine them.
Syntactic Analysis

Syntactic analysis identifies the hierarchical structure of an expression, which may be nested.

Each call to scheme_read consumes the input tokens for exactly one expression

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

**Base case:** symbols and numbers

**Recursive call:** scheme_read sub-expressions and combine them
Syntactic Analysis

Syntactic analysis identifies the hierarchical structure of an expression, which may be nested.

Each call to scheme_read consumes the input tokens for exactly one expression.

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

**Base case:** symbols and numbers

**Recursive call:** scheme_read sub-expressions and combine them
Syntactic Analysis

Syntactic analysis identifies the hierarchical structure of an expression, which may be nested.

Each call to scheme_read consumes the input tokens for exactly one expression.

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

**Base case:** symbols and numbers

**Recursive call:** scheme_read sub-expressions and combine them
Syntactic Analysis

Syntactic analysis identifies the hierarchical structure of an expression, which may be nested.

Each call to scheme_read consumes the input tokens for exactly one expression

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

**Base case:** symbols and numbers

**Recursive call:** scheme_read sub-expressions and combine them
Syntactic Analysis

Syntactic analysis identifies the hierarchical structure of an expression, which may be nested.

Each call to scheme_read consumes the input tokens for exactly one expression:

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

**Base case:** symbols and numbers

**Recursive call:** scheme_read sub-expressions and combine them.
Syntactic Analysis

Syntactic analysis identifies the hierarchical structure of an expression, which may be nested.

Each call to scheme_read consumes the input tokens for exactly one expression

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

**Base case:** symbols and numbers

**Recursive call:** scheme_read sub-expressions and combine them
Syntactic Analysis

Syntactic analysis identifies the hierarchical structure of an expression, which may be nested

Each call to scheme_read consumes the input tokens for exactly one expression

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

**Base case:** symbols and numbers

**Recursive call:** scheme_read sub-expressions and combine them
Syntactic Analysis

Syntactic analysis identifies the hierarchical structure of an expression, which may be nested.

Each call to scheme_read consumes the input tokens for exactly one expression:

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

**Base case:** symbols and numbers

**Recursive call:** scheme_read sub-expressions and combine them

(Demo)
Calculator

(Demo)
The Pair Class

The Pair class represents Scheme pairs and lists. A list is a pair whose second element is either a list or nil.
The Pair Class

The Pair class represents Scheme pairs and lists. A list is a pair whose second element is either a list or nil.

class Pair:
    """A Pair has two instance attributes: first and second.
    For a Pair to be a well-formed list, second is either a well-formed list or nil.
    Some methods only apply to well-formed lists.
    """
    def __init__(self, first, second):
        self.first = first
        self.second = second
The Pair Class

The Pair class represents Scheme pairs and lists. A list is a pair whose second element is either a list or nil.

class Pair:
    """A Pair has two instance attributes:
    first and second.

    For a Pair to be a well-formed list,
    second is either a well-formed list or nil.
    Some methods only apply to well-formed lists.
    """
    def __init__(self, first, second):
        self.first = first
        self.second = second

>>> s = Pair(1, Pair(2, Pair(3, nil)))
The Pair Class

The Pair class represents Scheme pairs and lists. A list is a pair whose second element is either a list or nil.

class Pair:
    """A Pair has two instance attributes: first and second.
    For a Pair to be a well-formed list, second is either a well-formed list or nil.
    Some methods only apply to well-formed lists.
    """
    def __init__(self, first, second):
        self.first = first
        self.second = second

>>> s = Pair(1, Pair(2, Pair(3, nil)))
>>> print(s)
(1 2 3)
The Pair Class

The Pair class represents Scheme pairs and lists. A list is a pair whose second element is either a list or nil.

class Pair:
    """A Pair has two instance attributes: first and second.

    For a Pair to be a well-formed list, second is either a well-formed list or nil.
    Some methods only apply to well-formed lists.
    """
    def __init__(self, first, second):
        self.first = first
        self.second = second

>>> s = Pair(1, Pair(2, Pair(3, nil)))
>>> print(s)
(1 2 3)
>>> len(s)
3
The Pair Class

The Pair class represents Scheme pairs and lists. A list is a pair whose second element is either a list or nil.

class Pair:
    """A Pair has two instance attributes:
    first and second.
    For a Pair to be a well-formed list,
    second is either a well-formed list or nil.
    Some methods only apply to well-formed lists.
    """
    def __init__(self, first, second):
        self.first = first
        self.second = second

>>> s = Pair(1, Pair(2, Pair(3, nil)))
>>> print(s)
(1 2 3)
>>> len(s)
3
>>> print(Pair(1, 2))
(1 . 2)
The Pair Class

The Pair class represents Scheme pairs and lists. A list is a pair whose second element is either a list or nil.

class Pair:
    """A Pair has two instance attributes: first and second.

For a Pair to be a well-formed list, second is either a well-formed list or nil. Some methods only apply to well-formed lists."
""
def __init__(self, first, second):
    self.first = first
    self.second = second

>>> s = Pair(1, Pair(2, Pair(3, nil)))
>>> print(s)
(1 2 3)
>>> len(s)
3
>>> print(Pair(1, 2))
(1 . 2)
>>> print(Pair(1, Pair(2, 3)))
(1 2 . 3)
The Pair Class

The Pair class represents Scheme pairs and lists. A list is a pair whose second element is either a list or nil.

class Pair:
    """A Pair has two instance attributes: first and second.
    For a Pair to be a well-formed list, second is either a well-formed list or nil.
    Some methods only apply to well-formed lists."

    def __init__(self, first, second):
        self.first = first
        self.second = second

>>> s = Pair(1, Pair(2, Pair(3, nil)))
>>> print(s)
(1 2 3)
>>> len(s)
3
>>> print(Pair(1, 2))
(1 . 2)
>>> print(Pair(1, Pair(2, 3)))
(1 2 . 3)
>>> len(Pair(1, Pair(2, 3)))
Traceback (most recent call last):
  ...
TypeError: length attempted on improper list
The Pair Class

The Pair class represents Scheme pairs and lists. A list is a pair whose second element is either a list or nil.

```python
class Pair:
    """A Pair has two instance attributes: first and second.
    
    For a Pair to be a well-formed list, second is either a well-formed list or nil.
    Some methods only apply to well-formed lists.
    """
    def __init__(self, first, second):
        self.first = first
        self.second = second

>>> s = Pair(1, Pair(2, Pair(3, nil)))
>>> print(s)
(1 2 3)
>>> len(s)
3
>>> print(Pair(1, 2))
(1 . 2)
>>> print(Pair(1, Pair(2, 3)))
(1 2 . 3)
>>> len(Pair(1, Pair(2, 3)))
Traceback (most recent call last):
...
TypeError: length attempted on improper list
```

Scheme expressions are represented as Scheme lists! Source code is data
The Pair Class

The Pair class represents Scheme pairs and lists. A list is a pair whose second element is either a list or nil.

```python
class Pair:
    """A Pair has two instance attributes:
    first and second.
    For a Pair to be a well-formed list,
    second is either a well-formed list or nil.
    Some methods only apply to well-formed lists.
    """
    def __init__(self, first, second):
        self.first = first
        self.second = second
```

Scheme expressions are represented as Scheme lists! Source code is data

(Demo)
Calculator Syntax
Calculator Syntax

The Calculator language has primitive expressions and call expressions. (That's it!)
Calculator Syntax

The Calculator language has primitive expressions and call expressions. (That's it!)

A primitive expression is a number: 2 -4 5.6
The Calculator language has primitive expressions and call expressions. (That's it!)

A primitive expression is a number: 2, −4, 5.6

A call expression is a combination that begins with an operator (+, −, *, /) followed by 0 or more expressions: (+ 1 2 3) (/ 3 (+ 4 5))
The Calculator language has primitive expressions and call expressions. (That's it!)

A primitive expression is a number: 2 -4 5.6

A call expression is a combination that begins with an operator (+, -, *, /) followed by 0 or more expressions: (+ 1 2 3) (/ 3 (+ 4 5))

Expressions are represented as Scheme lists (Pair instances) that encode tree structures.
The Calculator language has primitive expressions and call expressions. (That's it!)

A primitive expression is a number: 2 -4 5.6

A call expression is a combination that begins with an operator (+, -, *, /) followed by 0 or more expressions: (+ 1 2 3) (/ 3 (+ 4 5))

Expressions are represented as Scheme lists (Pair instances) that encode tree structures.

Expression

(* 3
  (+ 4 5)
  (* 6 7 8))
The Calculator language has primitive expressions and call expressions. (That's it!)

A primitive expression is a number: \(2\), \(-4\), \(5.6\).

A call expression is a combination that begins with an operator (+, -, *, /) followed by 0 or more expressions: \((+ 1 2 3)\), \((/ 3 (+ 4 5))\).

Expressions are represented as Scheme lists (Pair instances) that encode tree structures.
**Calculator Syntax**

The Calculator language has primitive expressions and call expressions. (That's it!)

A primitive expression is a number:  2   -4   5.6

A call expression is a combination that begins with an operator (+, -, *, /) followed by 0 or more expressions:  (+ 1 2 3)      (/ 3 (+ 4 5))

Expressions are represented as Scheme lists (Pair instances) that encode tree structures.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Expression Tree</th>
<th>Representation as Pairs</th>
</tr>
</thead>
<tbody>
<tr>
<td>(* 3</td>
<td><img src="image" alt="Expression Tree Diagram" /></td>
<td><img src="image" alt="Representation as Pairs Diagram" /></td>
</tr>
<tr>
<td>(+ 4 5)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(* 6 7 8))</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Calculator Semantics
Calculator Semantics

The value of a calculator expression is defined recursively.
Calculator Semantics

The value of a calculator expression is defined recursively.

**Primitive:** A number evaluates to itself.
**Calculator Semantics**

The value of a calculator expression is defined recursively.

**Primitive:** A number evaluates to itself.

**Call:** A call expression evaluates to its argument values combined by an operator.
Calculator Semantics

The value of a calculator expression is defined recursively.

**Primitive:** A number evaluates to itself.

**Call:** A call expression evaluates to its argument values combined by an operator.

+: Sum of the arguments
Calculator Semantics

The value of a calculator expression is defined recursively.

**Primitive**: A number evaluates to itself.

**Call**: A call expression evaluates to its argument values combined by an operator.

- `+`: Sum of the arguments
- `*`: Product of the arguments
Calculator Semantics

The value of a calculator expression is defined recursively.

**Primitive:** A number evaluates to itself.

**Call:** A call expression evaluates to its argument values combined by an operator.

- +: Sum of the arguments
- *: Product of the arguments
- -: If one argument, negate it. If more than one, subtract the rest from the first.
Calculator Semantics

The value of a calculator expression is defined recursively.

**Primitive**: A number evaluates to itself.

**Call**: A call expression evaluates to its argument values combined by an operator.
- `+`: Sum of the arguments
- `*`: Product of the arguments
- `-`: If one argument, negate it. If more than one, subtract the rest from the first.
- `/`: If one argument, invert it. If more than one, divide the rest from the first.
Calculator Semantics

The value of a calculator expression is defined recursively.

**Primitive**: A number evaluates to itself.

**Call**: A call expression evaluates to its argument values combined by an operator.
  
  +: Sum of the arguments
  *
  -: If one argument, negate it. If more than one, subtract the rest from the first.
  /: If one argument, invert it. If more than one, divide the rest from the first.

```
Expression
(+ 5
  (* 2 3)
  (* 2 5 5))
```
Calculator Semantics

The value of a calculator expression is defined recursively.

**Primitive**: A number evaluates to itself.

**Call**: A call expression evaluates to its argument values combined by an operator.
- `+`: Sum of the arguments
- `*`: Product of the arguments
- `-`: If one argument, negate it. If more than one, subtract the rest from the first.
- `/`: If one argument, invert it. If more than one, divide the rest from the first.

```
(+ 5 (* 2 3) (* 2 5 5))
```

Expression Tree:
```
+ 5  
* 2   
3  
* 2 5 5
```
Calculator Semantics

The value of a calculator expression is defined recursively.

**Primitive**: A number evaluates to itself.

**Call**: A call expression evaluates to its argument values combined by an operator.
- `+`: Sum of the arguments
- `*`: Product of the arguments
- `-`: If one argument, negate it. If more than one, subtract the rest from the first.
- `/`: If one argument, invert it. If more than one, divide the rest from the first.

Expression: 
```
(+ 5
  (* 2 3)
  (* 2 5 5))
```

Expression Tree:
```
+ 5
  6
  * 2 3
  * 2 5 5
```
Calculator Semantics

The value of a calculator expression is defined recursively.

**Primitive:** A number evaluates to itself.

**Call:** A call expression evaluates to its argument values combined by an operator.

+: Sum of the arguments

*.: Product of the arguments

-.: If one argument, negate it. If more than one, subtract the rest from the first.

/: If one argument, invert it. If more than one, divide the rest from the first.
Calculator Semantics

The value of a calculator expression is defined recursively.

**Primitive**: A number evaluates to itself.

**Call**: A call expression evaluates to its argument values combined by an operator.

- `+`: Sum of the arguments
- `*`: Product of the arguments
- `-`: If one argument, negate it. If more than one, subtract the rest from the first.
- `/`: If one argument, invert it. If more than one, divide the rest from the first.

```
Expression       Expression Tree

(+ 5
  (* 2 3)
  (* 2 5 5))
```

13 + 5 * 2 3 * 2 5 5 = 61
Evaluation
The Eval Function
The Eval Function

The eval function computes the value of an expression, which is always a number.
The Eval Function

The eval function computes the value of an expression, which is always a number.

It is a generic function that dispatches on the type of the expression (primitive or call).
The Eval Function

The eval function computes the value of an expression, which is always a number.

It is a generic function that dispatches on the type of the expression (primitive or call).

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Language Semantics</th>
</tr>
</thead>
</table>
The Eval Function

The eval function computes the value of an expression, which is always a number.

It is a generic function that dispatches on the type of the expression (primitive or call).

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Language Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>def calc_eval(exp):</td>
<td></td>
</tr>
<tr>
<td>if type(exp) in (int, float):</td>
<td></td>
</tr>
<tr>
<td>return exp</td>
<td></td>
</tr>
<tr>
<td>elif isinstance(exp, Pair):</td>
<td></td>
</tr>
<tr>
<td>arguments = exp.second.map(calc_eval)</td>
<td></td>
</tr>
<tr>
<td>return calc_apply(exp.first, arguments)</td>
<td></td>
</tr>
<tr>
<td>else:</td>
<td></td>
</tr>
<tr>
<td>raise TypeError</td>
<td></td>
</tr>
</tbody>
</table>
The Eval Function

The eval function computes the value of an expression, which is always a number.

It is a generic function that dispatches on the type of the expression (primitive or call).

### Implementation

```python
def calc_eval(exp):
    if type(exp) in (int, float):
        return exp
    elif isinstance(exp, Pair):
        arguments = exp.second.map(calc_eval)
        return calc_apply(exp.first, arguments)
    else:
        raise TypeError
```

### Language Semantics

*A number evaluates...*
The Eval Function

The eval function computes the value of an expression, which is always a number.

It is a generic function that dispatches on the type of the expression (primitive or call).

**Implementation**

```python
def calc_eval(exp):
    if type(exp) in (int, float):
        return exp
    elif isinstance(exp, Pair):
        arguments = exp.second.map(calc_eval)
        return calc_apply(exp.first, arguments)
    else:
        raise TypeError
```

**Language Semantics**

*A number evaluates...*  
*to itself*
The Eval Function

The eval function computes the value of an expression, which is always a number.

It is a generic function that dispatches on the type of the expression (primitive or call).

**Implementation**

```python
def calc_eval(exp):
    if type(exp) in (int, float):
        return exp
    elif isinstance(exp, Pair):
        arguments = exp.second.map(calc_eval)
        return calc_apply(exp.first, arguments)
    else:
        raise TypeError
```

**Language Semantics**

- A number evaluates... to itself
- A call expression evaluates...
The Eval Function

The eval function computes the value of an expression, which is always a number. It is a generic function that dispatches on the type of the expression (primitive or call).

### Implementation

```python
def calc_eval(exp):
    if type(exp) in (int, float):
        return exp
    elif isinstance(exp, Pair):
        arguments = exp.second.map(calc_eval)
        return calc_apply(exp.first, arguments)
    else:
        raise TypeError
```

### Language Semantics

- **A number evaluates...**
  - to itself
- **A call expression evaluates...**
  - to its argument values
The Eval Function

The eval function computes the value of an expression, which is always a number.

It is a generic function that dispatches on the type of the expression (primitive or call).

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Language Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>def calc_eval(exp):</code></td>
<td><strong>A number evaluates...</strong></td>
</tr>
<tr>
<td><code>if type(exp) in (int, float):</code></td>
<td><strong>to itself</strong></td>
</tr>
<tr>
<td><code>return exp</code></td>
<td><strong>A call expression evaluates...</strong></td>
</tr>
<tr>
<td><code>elif isinstance(exp, Pair):</code></td>
<td><strong>to its argument values</strong></td>
</tr>
<tr>
<td><code>arguments = exp.second.map(calc_eval)</code></td>
<td><strong>combined by an operator</strong></td>
</tr>
<tr>
<td><code>return calc_apply(exp.first, arguments)</code></td>
<td></td>
</tr>
<tr>
<td><code>else:</code></td>
<td></td>
</tr>
<tr>
<td><code>raise TypeError</code></td>
<td></td>
</tr>
</tbody>
</table>
The Eval Function

The eval function computes the value of an expression, which is always a number.

It is a generic function that dispatches on the type of the expression (primitive or call).

### Implementation

```python
def calc_eval(exp):
    if type(exp) in (int, float):
        return exp
    elif isinstance(exp, Pair):
        arguments = exp.second.map(calc_eval)
        return calc_apply(exp.first, arguments)
    else:
        raise TypeError
```

### Language Semantics

- **A number evaluates...**
  - to itself

- **A call expression evaluates...**
  - to its argument values
  - combined by an operator

Recursive call returns a number for each operand.
The Eval Function

The eval function computes the value of an expression, which is always a number.

It is a generic function that dispatches on the type of the expression (primitive or call).

Implementation

```python
def calc_eval(exp):
    if type(exp) in (int, float):
        return exp
    elif isinstance(exp, Pair):
        arguments = exp.second.map(calc_eval)
        return calc_apply(exp.first, arguments)
    else:
        raise TypeError
```

Language Semantics

A number evaluates... to itself

A call expression evaluates... to its argument values combined by an operator
The Eval Function

The eval function computes the value of an expression, which is always a number.

It is a generic function that dispatches on the type of the expression (primitive or call).

**Implementation**

```python
def calc_eval(exp):
    if type(exp) in (int, float):
        return exp
    elif isinstance(exp, Pair):
        arguments = exp.second.map(calc_eval)
        return calc_apply(exp.first, arguments)
    else:
        raise TypeError
```

**Language Semantics**

A number evaluates...
- to itself

A call expression evaluates...
- to its argument values
- combined by an operator
The Eval Function

The eval function computes the value of an expression, which is always a number.

It is a generic function that dispatches on the type of the expression (primitive or call).

### Implementation

```python
def calc_eval(exp):
    if type(exp) in (int, float):
        return exp
    elif isinstance(exp, Pair):
        arguments = exp.second.map(calc_eval)
        return calc_apply(exp.first, arguments)
    else:
        raise TypeError
```

### Language Semantics

**A number evaluates...**
- *to itself*

**A call expression evaluates...**
- *to its argument values combined by an operator*
Applying Built-in Operators
Applying Built-in Operators

The apply function applies some operation to a (Scheme) list of argument values
Applying Built-in Operators

The apply function applies some operation to a (Scheme) list of argument values.

In calculator, all operations are named by built-in operators: +, -, *, /
Applying Built-in Operators

The apply function applies some operation to a (Scheme) list of argument values.

In calculator, all operations are named by built-in operators: +, -, *, /
Applying Built-in Operators

The apply function applies some operation to a (Scheme) list of argument values.

In calculator, all operations are named by built-in operators: +, -, *, /

```
def calc_apply(operator, args):
    if operator == '+':
        return reduce(add, args, 0)
    elif operator == '-':
        ...
    elif operator == '*':
        ...
    elif operator == '/':
        ...
    else:
        raise TypeError
```
Applying Built-in Operators

The apply function applies some operation to a (Scheme) list of argument values.

In calculator, all operations are named by built-in operators: +, −, *, /

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Language Semantics</th>
</tr>
</thead>
</table>
| def calc_apply(operator, args):
  if operator == '+':
    return reduce(add, args, 0)
  elif operator == '-':
    ...
  elif operator == '*':
    ...
  elif operator == '/':
    ...
  else:
    raise TypeError |
| +:                  |                   |
Applying Built-in Operators

The apply function applies some operation to a (Scheme) list of argument values.

In calculator, all operations are named by built-in operators: +, -, *, /

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Language Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>def calc_apply(operator, args):</td>
<td></td>
</tr>
<tr>
<td>if operator == '+':</td>
<td>+:</td>
</tr>
<tr>
<td>return reduce(add, args, 0)</td>
<td><em>Sum of the arguments</em></td>
</tr>
<tr>
<td>elif operator == '-':</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>elif operator == '*':</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>elif operator == '/':</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>else:</td>
<td></td>
</tr>
<tr>
<td>raise TypeError</td>
<td></td>
</tr>
</tbody>
</table>
Applying Built-in Operators

The apply function applies some operation to a (Scheme) list of argument values.

In calculator, all operations are named by built-in operators: +, -, *, /

Implementation

```python
def calc_apply(operator, args):
    if operator == '+':
        return reduce(add, args, 0)
    elif operator == '-':
        ...
    elif operator == '*':
        ...
    elif operator == '/':
        ...
    else:
        raise TypeError
```

Language Semantics

+:

\( \text{Sum of the arguments} \)

-:
Applying Built-in Operators

The apply function applies some operation to a (Scheme) list of argument values.

In calculator, all operations are named by built-in operators: +, -, *, /

### Implementation

```python
def calc_apply(operator, args):
    if operator == '+':
        return reduce(add, args, 0)
    elif operator == '-':
        ...
    elif operator == '*':
        ...
    elif operator == '/':
        ...
    else:
        raise TypeError
```

### Language Semantics

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+:</td>
<td>Sum of the arguments</td>
</tr>
<tr>
<td>-:</td>
<td>...</td>
</tr>
<tr>
<td>*/</td>
<td>...</td>
</tr>
</tbody>
</table>
Interactive Interpreters
Read-Eval-Print Loop
Read-Eval-Print Loop

The user interface for many programming languages is an interactive interpreter.
Read-Eval-Print Loop

The user interface for many programming languages is an interactive interpreter

1. Print a prompt
The user interface for many programming languages is an interactive interpreter

1. Print a prompt
2. Read text input from the user
Read-Eval-Print Loop

The user interface for many programming languages is an interactive interpreter

1. Print a prompt
2. **Read** text input from the user
3. Parse the text input into an expression
Read-Eval-Print Loop

The user interface for many programming languages is an interactive interpreter.

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

The user interface for many programming languages is an interactive interpreter

1. Print a prompt
2. Read text input from the user
3. Parse the text input into an expression
4. Evaluate the expression
5. If any errors occur, report those errors, otherwise
Read-Eval-Print Loop

The user interface for many programming languages is an interactive interpreter

1. Print a prompt
2. **Read** text input from the user
3. Parse the text input into an expression
4. **Evaluate** the expression
5. If any errors occur, report those errors, otherwise
6. **Print** the value of the expression and repeat
Read-Eval-Print Loop

The user interface for many programming languages is an interactive interpreter

1. Print a prompt
2. **Read** text input from the user
3. Parse the text input into an expression
4. **Evaluate** the expression
5. If any errors occur, report those errors, otherwise
6. **Print** the value of the expression and repeat

(Demo)
Raising Exceptions
Raising Exceptions

Exceptions are raised within lexical analysis, syntactic analysis, eval, and apply
Raising Exceptions

Exceptions are raised within lexical analysis, syntactic analysis, eval, and apply

Example exceptions
Raising Exceptions

Exceptions are raised within lexical analysis, syntactic analysis, eval, and apply.

Example exceptions:
- **Lexical analysis**: The token 2.3.4 raises ValueError("invalid numeral")
Raising Exceptions

Exceptions are raised within lexical analysis, syntactic analysis, eval, and apply

Example exceptions

- **Lexical analysis**: The token 2.3.4 raises ValueError("invalid numeral")
- **Syntactic analysis**: An extra ) raises SyntaxError("unexpected token")
Raising Exceptions

Exceptions are raised within lexical analysis, syntactic analysis, eval, and apply.

Example exceptions

- **Lexical analysis:** The token 2.3.4 raises ValueError("invalid numeral")
- **Syntactic analysis:** An extra ) raises SyntaxError("unexpected token")
- **Eval:** An empty combination raises TypeError("() is not a number or call expression")
Raising Exceptions

Exceptions are raised within lexical analysis, syntactic analysis, eval, and apply

Example exceptions

- **Lexical analysis**: The token 2.3.4 raises ValueError("invalid numeral")
- **Syntactic analysis**: An extra ) raises SyntaxError("unexpected token")
- **Eval**: An empty combination raises TypeError("() is not a number or call expression")
- **Apply**: No arguments to - raises TypeError("- requires at least 1 argument")
Raising Exceptions

Exceptions are raised within lexical analysis, syntactic analysis, eval, and apply.

Example exceptions:
- **Lexical analysis:** The token 2.3.4 raises ValueError("invalid numeral")
- **Syntactic analysis:** An extra ) raises SyntaxError("unexpected token")
- **Eval:** An empty combination raises TypeError("() is not a number or call expression")
- **Apply:** No arguments to - raises TypeError("- requires at least 1 argument")

(Demo)
Handling Exceptions
Handling Exceptions

An interactive interpreter prints information about each error
Handling Exceptions

An interactive interpreter prints information about each error.

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

An interactive interpreter prints information about each error.

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.

(Demo)