Parsing

A Parser takes as input a string that contains an expression and returns an expression tree.
A Parser takes as input a string that contains an expression and returns an expression tree.
Parsing

A Parser takes as input a string that contains an expression and returns an expression tree

'sum(2, 2)'
A Parser takes as input a string that contains an expression and returns an expression tree.

- **String**: 'add(2, 2)'
- **Expression Tree**: Exp('add', [2, 2])
A Parser takes as input a string that contains an expression and returns an expression tree.

\[
\text{string} \quad \text{parser} \quad \text{expression tree} \quad \text{Evaluator} \quad \text{value}
\]

'add(2, 2)' \quad \text{Exp('add', [2, 2])} \quad 4
A Parser takes as input a string that contains an expression and returns an expression tree.

\[
\text{string: 'add(2, 2)'} \quad \text{parser: Exp('add', [2, 2])} \quad \text{Evaluator: 4}
\]
A Parser takes as input a string that contains an expression and returns an expression tree.

- **String**: `'add(2, 2)'`
- **Parser**: `Exp('add', [2, 2])`
- **Evaluator**: Value = 4
A Parser takes as input a string that contains an expression and returns an expression tree.

string: 'add(2, 2)'
parser: Exp('add', [2, 2])
expression tree
Evaluator: value 4

Eval
Apply
A Parser takes as input a string that contains an expression and returns an expression tree.

\[
\text{string: 'add(2, 2)'}
\]

\[
\text{parser: Exp('add', [2, 2])}
\]

\[
\text{Evaluator: value} = 4
\]

Apply a function to its arguments.
A Parser takes as input a string that contains an expression and returns an expression tree.

string → parser → expression tree → Evaluator → value

'add(2, 2)' → Exp('add', [2, 2]) → 4

Apply a function to its arguments
A Parser takes as input a string that contains an expression and returns an expression tree.
A Parser takes as input a string that contains an expression and returns an expression tree.

**Diagram:**
- **String:** 'add(2, 2)'
- **Parser:** `Exp('add', [2, 2])`
- **Evaluator:**
  - **Eval:**
  - **Apply:**
    - **Evaluate operands**
    - **Apply a function to its arguments**
  - **Value:** 4
A Parser takes as input a string that contains an expression and returns an expression tree.
A Parser takes as input a string that contains an expression and returns an expression tree.
Two-Stage Parsing
Two-Stage Parsing

Lexical analyzer: Analyzes an input string as a sequence of tokens, which are symbols and delimiters
Two-Stage Parsing

**Lexical analyzer:** Analyzes an input string as a sequence of tokens, which are symbols and delimiters

**Syntactic analyzer:** Analyzes a sequence of tokens as an expression tree, which typically includes call expressions
Two-Stage Parsing

**Lexical analyzer:** Analyzes an input string as a sequence of tokens, which are symbols and delimiters

**Syntactic analyzer:** Analyzes a sequence of tokens as an expression tree, which typically includes call expressions

```python
def calc_parse(line):
```

Wednesday, November 2, 2011
Two-Stage Parsing

**Lexical analyzer:** Analyzes an input string as a sequence of tokens, which are symbols and delimiters

**Syntactic analyzer:** Analyzes a sequence of tokens as an expression tree, which typically includes call expressions

```python
def calc_parse(line):
    """Parse a line of calculator input."""
```
Two-Stage Parsing

**Lexical analyzer:** Analyzes an input string as a sequence of tokens, which are symbols and delimiters

**Syntactic analyzer:** Analyzes a sequence of tokens as an expression tree, which typically includes call expressions

```python
def calc_parse(line):
    """Parse a line of calculator input.""
    tokens = tokenize(line)
```
Two-Stage Parsing

**Lexical analyzer:** Analyzes an input string as a sequence of tokens, which are symbols and delimiters.

**Syntactic analyzer:** Analyzes a sequence of tokens as an expression tree, which typically includes call expressions.

```python
def calc_parse(line):
    """Parse a line of calculator input."""
    tokens = tokenize(line)
```

Lexical analysis is also called *tokenization*.
Two-Stage Parsing

**Lexical analyzer:** Analyzes an input string as a sequence of tokens, which are symbols and delimiters

**Syntactic analyzer:** Analyzes a sequence of tokens as an expression tree, which typically includes call expressions

```python
def calc_parse(line):
    """Parse a line of calculator input."""
    tokens = tokenize(line)
    expression_tree = analyze(tokens)
```

Lexical analysis is also called *tokenization*
Parsing with Local State

**Lexical analyzer:** Creates a list of tokens

**Syntactic analyzer:** Consumes a list of tokens
Lexical analyzer: Creates a list of tokens

Syntactic analyzer: Consumes a list of tokens

```python
def calc_parse(line):
    """Parse a line of calculator input.""
    tokens = tokenize(line)
    expression_tree = analyze(tokens)
```

Lexical analysis is also called *tokenization*. 
Lexical analyzer: Creates a list of tokens

Syntactic analyzer: Consumes a list of tokens

```python
def calc_parse(line):
    """Parse a line of calculator input."""
    tokens = tokenize(line)
    expression_tree = analyze(tokens)
    if len(tokens) > 0:
        raise SyntaxError('Extra token(s)')
```

Lexical analysis is also called tokenization
Parsing with Local State

**Lexical analyzer:** Creates a list of tokens

**Syntactic analyzer:** Consumes a list of tokens

```python
def calc_parse(line):
    """Parse a line of calculator input."""
    tokens = tokenize(line)
    expression_tree = analyze(tokens)
    if len(tokens) > 0:
        raise SyntaxError('Extra token(s)')
    return expression_tree
```

Lexical analysis is also called **tokenization**.
Lexical Analysis (a.k.a., Tokenization)
Lexical Analysis (a.k.a., Tokenization)

Lexical analysis identifies symbols and delimiters in a string
Lexical Analysis (a.k.a., Tokenization)

Lexical analysis identifies symbols and delimiters in a string

**Symbol:** A sequence of characters with meaning, representing a name (a.k.a., identifier), literal value, or reserved word
Lexical Analysis (a.k.a., Tokenization)

Lexical analysis identifies symbols and delimiters in a string

**Symbol:** A sequence of characters with meaning, representing a name (a.k.a., identifier), literal value, or reserved word

**Delimiter:** A sequence of characters that serves to define the syntactic structure of an expression
Lexical Analysis (a.k.a., Tokenization)

Lexical analysis identifies symbols and delimiters in a string

**Symbol:** A sequence of characters with meaning, representing a name (a.k.a., identifier), literal value, or reserved word

**Delimiter:** A sequence of characters that serves to define the syntactic structure of an expression

```python
>>> tokenize('add(2, mul(4, 6))')
['add', '(', '2', ',', 'mul', '(', '4', ',', '6', ')', ', ')']
```
Lexical Analysis (a.k.a., Tokenization)

Lexical analysis identifies symbols and delimiters in a string

Symbol: A sequence of characters with meaning, representing a name (a.k.a., identifier), literal value, or reserved word

Delimiter: A sequence of characters that serves to define the syntactic structure of an expression

>>> tokenize('add(2, mul(4, 6))')
['add', '(', '2', ',', 'mul', '(', '4', ',', '6', ')', ')']

Symbol: a built-in operator name
Lexical Analysis (a.k.a., Tokenization)

Lexical analysis identifies symbols and delimiters in a string

**Symbol**: A sequence of characters with meaning, representing a name (a.k.a., identifier), literal value, or reserved word

**Delimiter**: A sequence of characters that serves to define the syntactic structure of an expression

```python
>>> tokenize('add(2, mul(4, 6))')
['add', '(', '2', ',', 'mul', '(', '4', ',', '6', ')', ')']
```

Symbol: a built-in operator name

Delimiter
Lexical Analysis (a.k.a., Tokenization)

Lexical analysis identifies symbols and delimiters in a string

**Symbol:** A sequence of characters with meaning, representing a name (a.k.a., identifier), literal value, or reserved word

**Delimiter:** A sequence of characters that serves to define the syntactic structure of an expression

```python
>>> tokenize('add(2, mul(4, 6))')
['add', '(', '2', ',', 'mul', '(', '4', ',', '6', ')', ', ')']
```

- **Symbol:** a built-in operator name
- **Delimiter**
- **Symbol:** a literal
Lexical Analysis (a.k.a., Tokenization)

Lexical analysis identifies symbols and delimiters in a string

**Symbol:** A sequence of characters with meaning, representing a name (a.k.a., identifier), literal value, or reserved word

**Delimiter:** A sequence of characters that serves to define the syntactic structure of an expression

```python
>>> tokenize('add(2, mul(4, 6))')
['add', '(', '2', ',', 'mul', '(', '4', ',', '6', ')', ')']
```

Symbol: a built-in operator name  Delimiter  Symbol: a literal  Delimiter
Lexical Analysis (a.k.a., Tokenization)

Lexical analysis identifies symbols and delimiters in a string

Symbol: A sequence of characters with meaning, representing a name (a.k.a., identifier), literal value, or reserved word

Delimiter: A sequence of characters that serves to define the syntactic structure of an expression

```python
>>> tokenize('add(2, mul(4, 6))')
['add', '(', '2', ',', 'mul', '(', '4', ',', '6', ')', ')']
```

Symbol: a built-in operator name  
Delimiter  
Symbol: a literal  
Delimiter

(When viewed as a list of Calculator tokens)
Lexical Analysis By Inserting Spaces
Lexical Analysis By Inserting Spaces

Most lexical analyzers will explicitly inspect each character of the input string.
Lexical Analysis By Inserting Spaces

Most lexical analyzers will explicitly inspect each character of the input string.

For the syntax of Calculator, injecting white space suffices.
Lexical Analysis By Inserting Spaces

Most lexical analyzers will explicitly inspect each character of the input string.

For the syntax of Calculator, injecting white space suffices

```python
def tokenize(line):
```
Lexical Analysis By Inserting Spaces

Most lexical analyzers will explicitly inspect each character of the input string

For the syntax of Calculator, injecting white space suffices

```python
def tokenize(line):
    """Convert a string into a list of tokens."""
```
Lexical Analysis By Inserting Spaces

Most lexical analyzers will explicitly inspect each character of the input string

For the syntax of Calculator, injecting white space suffices

```python
def tokenize(line):
    """Convert a string into a list of tokens."""
    spaced = line.replace('(',' ( ').
Lexical Analysis By Inserting Spaces

Most lexical analyzers will explicitly inspect each character of the input string.

For the syntax of Calculator, injecting white space suffices.

```python
def tokenize(line):
    """Convert a string into a list of tokens.""
    spaced = line.replace('(',' ( ').replace(')', ' ) ')
```

Lexical Analysis By Inserting Spaces

Most lexical analyzers will explicitly inspect each character of the input string.

For the syntax of Calculator, injecting white space suffices.

```python
def tokenize(line):
    """Convert a string into a list of tokens.""
    spaced = line.replace('(',' ( ').
    spaced = spaced.replace(')', ', ')' )
    spaced = spaced.replace(',', ', ', ', ')
Lexical Analysis By Inserting Spaces

Most lexical analyzers will explicitly inspect each character of the input string

For the syntax of Calculator, injecting white space suffices

```python
def tokenize(line):
    """Convert a string into a list of tokens."""
    spaced = line.replace('(',' ( ').
    spaced = spaced.replace(')', ' ) ')
    spaced = spaced.replace(',,', ', ')
    return spaced.strip().split()
```
Lexical Analysis By Inserting Spaces

Most lexical analyzers will explicitly inspect each character of the input string.

For the syntax of Calculator, injecting white space suffices.

```python
def tokenize(line):
    """Convert a string into a list of tokens.""
    spaced = line.replace('(',' ( ').
    spaced = spaced.replace(')',', ' ) ')
    spaced = spaced.replace(',',', ', ', ')
    return spaced.strip().split()
```

Discard preceding or following white space
Lexical Analysis By Inserting Spaces

Most lexical analyzers will explicitly inspect each character of the input string

For the syntax of Calculator, injecting white space suffices

def tokenize(line):
    """Convert a string into a list of tokens.""
    spaced = line.replace('(',' ( ').
    spaced = spaced.replace(')', ' ) ')
    spaced = spaced.replace(',',' , ')
    return spaced.strip().split()

Discard preceding or following white space

Return a list of strings separated by white space
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 analyze consumes input tokens for an expression
Syntactic Analysis

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

Each call to analyze consumes input tokens for an expression.

```python
>>> tokens = tokenize('add(2, mul(4, 6))')
```
Syntactic Analysis

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

Each call to analyze consumes input tokens for an expression

```python
>>> tokens = tokenize('add(2, mul(4, 6))')
>>> tokens
['add', '(','2',',','mul', '(','4',',','6',')',')']
```
Syntactic Analysis

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

Each call to analyze consumes input tokens for an expression

```python
>>> tokens = tokenize('add(2, mul(4, 6))')

>>> tokens
given: ['add', '(', '2', ',', 'mul', '(', '4', ',', '6', ')', ')']

>>> analyze(tokens)
result: Exp('add', [2, Exp('mul', [4, 6])])
```
Syntactic Analysis

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

Each call to analyze consumes input tokens for an expression

```python
>>> tokens = tokenize('add(2, mul(4, 6))')

>>> tokens
['add', '(','2',',','mul', '(','4',',','6',')',')']

>>> analyze(tokens)
Exp('add', [2, Exp('mul', [4, 6])])

>>> tokens
[]
```
Recursive Syntactic Analysis
Recursive Syntactic Analysis

A predictive recursive descent parser inspects only \( k \) tokens to decide how to proceed, for some fixed \( k \).
Recursive Syntactic Analysis

A predictive recursive descent parser inspects only $k$ tokens to decide how to proceed, for some fixed $k$.

*Can English be parsed via predictive recursive descent?*
Recursive Syntactic Analysis

A predictive recursive descent parser inspects only $k$ tokens to decide how to proceed, for some fixed $k$.

*Can English be parsed via predictive recursive descent?*

The horse raced past the barn fell.
Recursive Syntactic Analysis

A predictive recursive descent parser inspects only $k$ tokens to decide how to proceed, for some fixed $k$.

*Can English be parsed via predictive recursive descent?*

The horse *raced* past the barn *fell*.  

*ridden*
Recursive Syntactic Analysis

A predictive recursive descent parser inspects only $k$ tokens to decide how to proceed, for some fixed $k$.

Can English be parsed via predictive recursive descent?

The horse --raced-- past the barn fell.

(ridden)
Recursive Syntactic Analysis

A predictive recursive descent parser inspects only $k$ tokens to decide how to proceed, for some fixed $k$.

Can English be parsed via predictive recursive descent?

The horse raced past the barn fell.

(sentence subject)

(ridden)

(that was)
Recursive Syntactic Analysis

A predictive recursive descent parser inspects only \( k \) tokens to decide how to proceed, for some fixed \( k \).

*Can English be parsed via predictive recursive descent?*

The horse raced past the barn fell.

Sentence subject

The horse (that was ridden) past the barn fell.

You got Gardenpath'd!
Recursive Syntactic Analysis

A predictive recursive descent parser inspects only $k$ tokens to decide how to proceed, for some fixed $k$. 
Recursive Syntactic Analysis

A predictive recursive descent parser inspects only \( k \) tokens to decide how to proceed, for some fixed \( k \).

```python
def analyze(tokens):
```

Wednesday, November 2, 2011
Recursive Syntactic Analysis

A predictive recursive descent parser inspects only $k$ tokens to decide how to proceed, for some fixed $k$.

```python
def analyze(tokens):
    token = analyze_token(tokens.pop(0))
```
Recursive Syntactic Analysis

A predictive recursive descent parser inspects only $k$ tokens to decide how to proceed, for some fixed $k$.

```python
def analyze(tokens):
    token = analyze_token(tokens.pop(0))
```

In Calculator, we inspect 1 token.
Recursive Syntactic Analysis

A predictive recursive descent parser inspects only \( k \) tokens to decide how to proceed, for some fixed \( k \).

```python
def analyze(tokens):
    token = analyze_token(tokens.pop(0))
```

Coerces numeric symbols to numeric values

In Calculator, we inspect 1 token
Recursive Syntactic Analysis

A predictive recursive descent parser inspects only \( k \) tokens to decide how to proceed, for some fixed \( k \).

```python
def analyze(tokens):
    token = analyze_token(tokens.pop(0))
    if type(token) in (int, float):
        Coerces numeric symbols to numeric values
        In Calculator, we inspect 1 token
```

Wednesday, November 2, 2011
Recursive Syntactic Analysis

A predictive recursive descent parser inspects only k tokens to decide how to proceed, for some fixed k.

```python
def analyze(tokens):
    token = analyze_token(tokens.pop(0))
    if type(token) in (int, float):
        return token
```

In Calculator, we inspect 1 token

Coerces numeric symbols to numeric values
Recursive Syntactic Analysis

A predictive recursive descent parser inspects only $k$ tokens to decide how to proceed, for some fixed $k$.

```python
def analyze(tokens):
    token = analyze_token(tokens.pop(0))
    if type(token) in (int, float):
        return token
```

- Coerces numeric symbols to numeric values
- In Calculator, we inspect 1 token
- Numbers are complete expressions
Recursive Syntactic Analysis

A predictive recursive descent parser inspects only \( k \) tokens to decide how to proceed, for some fixed \( k \).

```python
def analyze(tokens):
    token = analyze_token(tokens.pop(0))
    if type(token) in (int, float):
        return token
    else:
        In Calculator, we inspect 1 token

Coerces numeric symbols to numeric values

Numbers are complete expressions
```
Recursive Syntactic Analysis

A predictive recursive descent parser inspects only $k$ tokens to decide how to proceed, for some fixed $k$.

```python
def analyze(tokens):
    token = analyze_token(tokens.pop(0))
    if type(token) in (int, float):
        return token
    else:
        tokens.pop(0)  # Remove ( (Calculator)
```

- Coerces numeric symbols to numeric values
- In Calculator, we inspect 1 token
- Numbers are complete expressions
A predictive recursive descent parser inspects only \( k \) tokens to decide how to proceed, for some fixed \( k \).

```python
def analyze(tokens):
    token = analyze_token(tokens.pop(0))
    if type(token) in (int, float):
        return token
    else:
        tokens.pop(0)  # Remove (  
        return Exp(token, analyze_operands(tokens))
```

In Calculator, we inspect 1 token. Numbers are complete expressions.

Coerces numeric symbols to numeric values.
A predictive recursive descent parser inspects only $k$ tokens to decide how to proceed, for some fixed $k$.

```python
def analyze(tokens):
    token = analyze_token(tokens.pop(0))
    if type(token) in (int, float):
        return token
    else:
        tokens.pop(0)  # Remove (  
        return Exp(token, analyze_operands(tokens))
```

Coerces numeric symbols to numeric values

In Calculator, we inspect 1 token

Numbers are complete expressions

tokens no longer includes first two elements
Mutual Recursion in Analyze
def analyze(tokens):
    token = analyze_token(tokens.pop(0))
    if type(token) in (int, float):
        return token
    else:
        tokens.pop(0)  # Remove (  
        return Exp(token, analyze_operands(tokens))
Mutual Recursion in Analyze

```python
def analyze(tokens):
    token = analyze_token(tokens.pop(0))
    if type(token) in (int, float):
        return token
    else:
        tokens.pop(0)  # Remove (  
        return Exp(token, analyze_operands(tokens))

def analyze_operands(tokens):
```

Wednesday, November 2, 2011
def analyze(tokens):
    token = analyze_token(tokens.pop(0))
    if type(token) in (int, float):
        return token
    else:
        tokens.pop(0)  # Remove (  
        return Exp(token, analyze_operands(tokens))

def analyze_operands(tokens):
    operands = []
def analyze(tokens):
    token = analyze_token(tokens.pop(0))
    if type(token) in (int, float):
        return token
    else:
        tokens.pop(0)  # Remove (  
        return Exp(token, analyze_operands(tokens))

def analyze_operands(tokens):
    operands = []
    while tokens[0] != '):
Mutual Recursion in Analyze

```python
def analyze(tokens):
    token = analyze_token(tokens.pop(0))
    if type(token) in (int, float):
        return token
    else:
        tokens.pop(0)  # Remove (
        return Exp(token, analyze_operands(tokens))

def analyze_operands(tokens):
    operands = []
    while tokens[0] != ')':
        if operands:
```
Mutual Recursion in Analyze

```python
def analyze(tokens):
    token = analyze_token(tokens.pop(0))
    if type(token) in (int, float):
        return token
    else:
        tokens.pop(0)  # Remove (  
        return Exp(token, analyze_operands(tokens))

def analyze_operands(tokens):
    operands = []
    while tokens[0] != ')':
        if operands:
            tokens.pop(0)  # Remove ,
```

Wednesday, November 2, 2011
def analyze(tokens):
    token = analyze_token(tokens.pop(0))
    if type(token) in (int, float):
        return token
    else:
        tokens.pop(0) # Remove (  
        return Exp(token, analyze_operands(tokens))

def analyze_operands(tokens):
    operands = []
    while tokens[0] != ')':
        if operands:
            tokens.pop(0) # Remove ,
            operands.append(analyze(tokens))
def analyze(tokens):
    token = analyze_token(tokens.pop(0))
    if type(token) in (int, float):
        return token
    else:
        tokens.pop(0)  # Remove (
        return Exp(token, analyze_operands(tokens))

def analyze_operands(tokens):
    operands = []
    while tokens[0] != ')':
        if operands:
            tokens.pop(0)  # Remove ,
            operands.append(analyze(tokens))
        tokens.pop(0)  # Remove )
Mutual Recursion in Analyze

def analyze(tokens):
    token = analyze_token(tokens.pop(0))
    if type(token) in (int, float):
        return token
    else:
        tokens.pop(0)  # Remove (  
        return Exp(token, analyze_operands(tokens))

def analyze_operands(tokens):
    operands = []
    while tokens[0] != '):
        if operands:
            tokens.pop(0)  # Remove ,  
            operands.append(analyze(tokens))  
            tokens.pop(0)  # Remove )  
    return operands
def analyze(tokens):
    token = analyze_token(tokens.pop(0))
    if type(token) in (int, float):
        return token
    else:
        tokens.pop(0)  # Remove (
        return Exp(token, analyze_operands(tokens))

def analyze_operands(tokens):
    operands = []
    while tokens[0] != ')
        if operands:
            tokens.pop(0)  # Remove ,
            operands.append(analyze(tokens))
        tokens.pop(0)  # Remove )
    return operands
def analyze(tokens):
    token = analyze_token(tokens.pop(0))
    if type(token) in (int, float):
        return token
    else:
        tokens.pop(0)  # Remove (
        return Exp(token, analyze_operands(tokens))

def analyze_operands(tokens):
    operands = []
    while tokens[0] != ')':
        if operands:
            tokens.pop(0)  # Remove ,
            operands.append(analyze(tokens))
        tokens.pop(0)  # Remove )
    return operands
Mutual Recursion in Analyze

```python
def analyze(tokens):
    token = analyze_token(tokens.pop(0))
    if type(token) in (int, float):
        return token
    else:
        tokens.pop(0)  # Remove (  
        return Exp(token, analyze_operands(tokens))

def analyze_operands(tokens):
    operands = []
    while tokens[0] != '):
        if operands:
            tokens.pop(0)  # Remove ,
            operands.append(analyze(tokens))
        tokens.pop(0)  # Remove )
    return operands
```

['add', '(, '2', ',', '3', ')']
def analyze(tokens):
    token = analyze_token(tokens.pop(0))
    if type(token) in (int, float):
        return token
    else:
        tokens.pop(0)  # Remove (  
        return Exp(token, analyze_operands(tokens))

def analyze_operands(tokens):
    operands = []
    while tokens[0] != '):
        if operands:
            tokens.pop(0)  # Remove ,
            operands.append(analyze(token))
        tokens.pop(0)  # Remove )
    return operands
def analyze(tokens):
    token = analyze_token(tokens.pop(0))
    if type(token) in (int, float):
        return token
    else:
        tokens.pop(0)  # Remove (
        return Exp(token, analyze_operands(tokens))

def analyze_operands(tokens):
    operands = []
    while tokens[0] != ')':
        if operands:
            tokens.pop(0)  # Remove ,
            operands.append(analyze(tokens))
        tokens.pop(0)  # Remove )
    return operands
Mutual Recursion in Analyze

```python
def analyze(tokens):
    token = analyze_token(tokens.pop(0))
    if type(token) in (int, float):
        return token
    else:
        tokens.pop(0)  # Remove (  
        return Exp(token, analyze_operands(tokens))
```

```python
def analyze_operands(tokens):
    operands = []
    while tokens[0] != ')':
        if operands:
            tokens.pop(0)  # Remove ,  
            operands.append(analyze(tokens))
        tokens.pop(0)  # Remove )
    return operands
```
Mutual Recursion in Analyze

```python
def analyze(tokens):
    token = analyze_token(tokens.pop(0))
    if type(token) in (int, float):
        return token
    else:
        tokens.pop(0)  # Remove (  
        return Exp(token, analyze_operands(tokens))

def analyze_operands(tokens):
    operands = []
    while tokens[0] != ')':
        if operands:
            tokens.pop(0)  # Remove ,  
            operands.append(analyze(tokens))
        tokens.pop(0)  # Remove )
    return operands
```

Pass 1
Mutual Recursion in Analyze

```python
def analyze(tokens):
    token = analyze_token(tokens.pop(0))
    if type(token) in (int, float):
        return token
    else:
        tokens.pop(0)  # Remove (  
        return Exp(token, analyze_operands(tokens))

def analyze_operands(tokens):
    operands = []
    while tokens[0] != '):
        if operands:
            tokens.pop(0)  # Remove ,
            operands.append(analyze(token))
        tokens.pop(0)  # Remove )
    return operands
```

Pass 1

[',', '3', ')']

Wednesday, November 2, 2011
Mutual Recursion in Analyze

```python
def analyze(tokens):
    token = analyze_token(tokens.pop(0))
    if type(token) in (int, float):
        return token
    else:
        tokens.pop(0)  # Remove (  
        return Exp(token, analyze_operands(tokens))

def analyze_operands(tokens):
    operands = []
    while tokens[0] != '):
        if operands:
            tokens.pop(0)  # Remove ,  
            operands.append(analyze_operations(tokens))
        tokens.pop(0)  # Remove )  
    return operands
```

Pass 1

```python
['add', '(', '2', ',', '3', ',')]
```

Pass 2

```python
def analyze(tokens):
```
Mutual Recursion in Analyze

```python
def analyze(tokens):
    token = analyze_token(tokens.pop(0))
    if type(token) in (int, float):
        return token
    else:
        tokens.pop(0)  # Remove (  
        return Exp(token, analyze_operands(tokens))

def analyze_operands(tokens):
    operands = []
    while tokens[0] != '):
        if operands:
            tokens.pop(0)  # Remove ,  
            operands.append(analyze(tokens))
        tokens.pop(0)  # Remove )
    return operands
```

Pass 1

`['add', '(', '2', ',', '3', ',')']`

Pass 2

```python
['2', ',', '3', ',')]
```

Wednesday, November 2, 2011
Mutual Recursion in Analyze

```
[ 'add', '(', '2', ',', ',', '3', ',') ]
[ '(', '2', ',', ',', '3', ',') ] def analyze(tokens):
    token = analyze_token(tokens.pop(0))
    if type(token) in (int, float):
        return token
    else:
        tokens.pop(0)  # Remove (  
        return Exp(token, analyze_operands(tokens))

[ '2', ',', ',', '3', ',') ]
[ '2', ',', ',', '3', ',') ] def analyze_operands(tokens):
    operands = []
    while tokens[0] != '):
        if operands:
            tokens.pop(0)  # Remove ,
            operands.append(analyze(tokens))
        tokens.pop(0)  # Remove )
    return operands
```

Pass 1  Pass 2

```
[ ', , '3', ,') ]
[ '3', ,') ]
[ '', '3', ',') ]
[ '3', ,') ]
```
def analyze(tokens):
    token = analyze_token(tokens.pop(0))
    if type(token) in (int, float):
        return token
    else:
        tokens.pop(0)  # Remove (
        return Exp(token, analyze_operands(tokens))

def analyze_operands(tokens):
    operands = []
    while tokens[0] != '):
        if operands:
            tokens.pop(0)  # Remove ,
            operands.append(analyze(token))
        tokens.pop(0)  # Remove )
    return operands
Token Coercion
Token Coercion

Parsers typically identify the form of each expression, so that eval can dispatch on that form.
Token Coercion

Parsers typically identify the form of each expression, so that eval can dispatch on that form.

In Calculator, the form is determined by the expression type.
Token Coercion

Parsers typically identify the form of each expression, so that eval can dispatch on that form.

In Calculator, the form is determined by the expression type:
- Primitive expressions are int or float values.
Token Coercion

Parsers typically identify the form of each expression, so that eval can dispatch on that form

In Calculator, the form is determined by the expression type
- Primitive expressions are int or float values
- Call expressions are Exp instances
Token Coercion

Parsers typically identify the form of each expression, so that eval can dispatch on that form.

In Calculator, the form is determined by the expression type:
- Primitive expressions are int or float values
- Call expressions are Exp instances

```python
def analyze_token(token):
```
Token Coercion

Parsers typically identify the form of each expression, so that eval can dispatch on that form

In Calculator, the form is determined by the expression type
- Primitive expressions are int or float values
- Call expressions are Exp instances

```python
def analyze_token(token):
    try:
```
Token Coercion

Parsers typically identify the form of each expression, so that eval can dispatch on that form.

In Calculator, the form is determined by the expression type:

- Primitive expressions are int or float values
- Call expressions are Exp instances

```python
def analyze_token(token):
    try:
        return int(token)
```
Token Coercion

Parsers typically identify the form of each expression, so that eval can dispatch on that form.

In Calculator, the form is determined by the expression type:
- Primitive expressions are int or float values
- Call expressions are Exp instances

```python
def analyze_token(token):
    try:
        return int(token)
    except (TypeError, ValueError):
```

Wednesday, November 2, 2011
Token Coercion

Parsers typically identify the form of each expression, so that eval can dispatch on that form.

In Calculator, the form is determined by the expression type:
- Primitive expressions are int or float values
- Call expressions are Exp instances

```python
def analyze_token(token):
    try:
        return int(token)
    except (TypeError, ValueError):
        try:
```

Wednesday, November 2, 2011
Token Coercion

Parsers typically identify the form of each expression, so that eval can dispatch on that form

In Calculator, the form is determined by the expression type
• Primitive expressions are int or float values
• Call expressions are Exp instances

```python
def analyze_token(token):
    try:
        return int(token)
    except (TypeError, ValueError):
        try:
            return float(token)
```
Token Coercion

Parsers typically identify the form of each expression, so that eval can dispatch on that form

In Calculator, the form is determined by the expression type

- Primitive expressions are int or float values
- Call expressions are Exp instances

```python
def analyze_token(token):
    try:
        return int(token)
    except (TypeError, ValueError):
        try:
            return float(token)
        except (TypeError, ValueError):
```

Wednesday, November 2, 2011
Token Coercion

Parsers typically identify the form of each expression, so that eval can dispatch on that form

In Calculator, the form is determined by the expression type

- Primitive expressions are int or float values
- Call expressions are Exp instances

```python
def analyze_token(token):
    try:
        return int(token)
    except (TypeError, ValueError):
        try:
            return float(token)
        except (TypeError, ValueError):
            return token
```
Token Coercion

Parsers typically identify the form of each expression, so that eval can dispatch on that form

In Calculator, the form is determined by the expression type
- Primitive expressions are int or float values
- Call expressions are Exp instances

```python
def analyze_token(token):
    try:
        return int(token)
    except (TypeError, ValueError):
        try:
            return float(token)
        except (TypeError, ValueError):
            return token
```

What would change if we deleted this?
Error Handling: Analyze
known_operators = ['add', 'sub', 'mul', 'div', '+', '-', '*', '/']
Error Handling: Analyze

known_operators = ['add', 'sub', 'mul', 'div', '+', '-', '*', '/']

def analyze(tokens):

Error Handling: Analyze

known_operators = ['add', 'sub', 'mul', 'div', '+', '-', '*', '/']

def analyze(tokens):
    assert_non_empty(tokens)
Error Handling: Analyze

known_operators = ['add', 'sub', 'mul', 'div', '+', '-', '*', '/']

def analyze(tokens):
    assert_non_empty(tokens)
    token = analyze_token(tokens.pop(0))
known_operators = ['add', 'sub', 'mul', 'div', '+', '-', '*', '/']

def analyze(tokens):
    assert_non_empty(tokens)
    token = analyze_token(tokens.pop(0))
    if type(token) in (int, float):
        return token
known_operators = ['add', 'sub', 'mul', 'div', '+', '-', '*', '/']

def analyze(tokens):
    assert_non_empty(tokens)
    token = analyze_token(tokens.pop(0))
    if type(token) in (int, float):
        return token
    if token in known_operators:
known_operators = ['add', 'sub', 'mul', 'div', '+', '-', '*', '/']

def analyze(tokens):
    assert_non_empty(tokens)
    token = analyze_token(tokens.pop(0))
    if type(token) in (int, float):
        return token
    if token in known_operators:
        if len(tokens) == 0 or tokens.pop(0) != '(':  
            raise SyntaxError('expected ( after ' + token)
known_operators = ['add', 'sub', 'mul', 'div', '+', '-', '*', '/']

def analyze(tokens):
    assert_non_empty(tokens)
    token = analyze_token(tokens.pop(0))
    if type(token) in (int, float):
        return token
    if token in known_operators:
        if len(tokens) == 0 or tokens.pop(0) != '(
            raise SyntaxError('expected ( after ' + token)
        return Exp(token, analyze_operands(tokens))
known_operators = ['add', 'sub', 'mul', 'div', '+', '-', '*', '/']

def analyze(tokens):
    assert_non_empty(tokens)
    token = analyze_token(tokens.pop(0))
    if type(token) in (int, float):
        return token
    if token in known_operators:
        if len(tokens) == 0 or tokens.pop(0) != '(':  # if len(tokens) == 0 or tokens[0] != '(':
            raise SyntaxError('expected ( after ' + token)
        return Exp(token, analyze_operands(tokens))
    else:
        raise SyntaxError('unexpected ' + token)
Error Handling: Analyze Operands
def analyze_operands(tokens):
def analyze_operands(tokens):
    assert_non_empty(tokens)
def analyze_operands(tokens):
    assert_non_empty(tokens)
    operands = []
def analyze_operands(tokens):
    assert_non_empty(tokens)
    operands = []
    while tokens[0] != '):

def analyze_operands(tokens):
    assert_non_empty(tokens)
    operands = []
    while tokens[0] != '):
        if operands and tokens.pop(0) != ',', ':
            raise SyntaxError('expected ,')
def analyze_operands(tokens):
    assert_non_empty(tokens)
    operands = []
    while tokens[0] != '):
        if operands and tokens.pop(0) != ',,':
            raise SyntaxError('expected ,')
        operands.append(analyze(tokens))
def analyze_operands(tokens):
    assert_non_empty(tokens)
    operands = []
    while tokens[0] != '):
        if operands and tokens.pop(0) != ',',:
            raise SyntaxError('expected ,')
        operands.append(analyze(tokens))
    assert_non_empty(tokens)
def analyze_operands(tokens):
    
    assert_non_empty(tokens)

    operands = []

    while tokens[0] != ')':
        
        if operands and tokens.pop(0) != ',', ':
            raise SyntaxError('expected ,,

        operands.append(analyze(tokens))

        assert_non_empty(tokens)

    tokens.pop(0)  # Remove )
def analyze_operands(tokens):
    assert_non_empty(tokens)
    operands = []
    while tokens[0] != '):
        if operands and tokens.pop(0) != ',', :
            raise SyntaxError('expected ,')
        operands.append(analyze(tokens))
        assert_non_empty(tokens)
    tokens.pop(0)  # Remove )
    return elements
def analyze_operands(tokens):
    assert_non_empty(tokens)
    operands = []
    while tokens[0] != ')':
        if operands and tokens.pop(0) != ',', ':
            raise SyntaxError('expected ,')
        operands.append(analyze(tokens))
        assert_non_empty(tokens)
    tokens.pop(0)  # Remove )
    return elements

def assert_non_empty(tokens):
    """Raise an exception if tokens is empty."""
    if len(tokens) == 0:
        raise SyntaxError('unexpected end of line')
Let's Break the Calculator
Let's Break the Calculator

I delete a statement that raises an exception
Let's Break the Calculator

I delete a statement that raises an exception.

You find an input that will crash Calculator.