61A Lecture 25

Monday, November 4

Announcements

Announcements

- Homework 7 due Tuesday 11/5 @ 11:59pm.

Announcements

- Homework 7 due Tuesday 11/5 @ 11:59pm.
- Project 1 composition revisions due Thursday 11/7 @ 11:59pm.

Announcements

- Homework 7 due Tuesday 11/5 @ 11:59pm.
- Project 1 composition revisions due Thursday 11/7 @ 11:59pm.
  - Instructions are posted on the course website (submit proj1revision)

Announcements

- Homework 7 due Tuesday 11/5 @ 11:59pm.
- Project 1 composition revisions due Thursday 11/7 @ 11:59pm.
    - Instructions are posted on the course website (submit proj1revision)
- Homework 8 due Tuesday 11/12 @ 11:59pm.

Announcements

- Homework 7 due Tuesday 11/5 @ 11:59pm.
- Project 1 composition revisions due Thursday 11/7 @ 11:59pm.
  - Instructions are posted on the course website (submit proj1revision)
- Homework 8 due Tuesday 11/12 @ 11:59pm.
  - All problems must be solved in Scheme

Announcements

- Homework 7 due Tuesday 11/5 @ 11:59pm.
- Project 1 composition revisions due Thursday 11/7 @ 11:59pm.
  - Instructions are posted on the course website (submit proj1revision)
- Homework 8 due Tuesday 11/12 @ 11:59pm.
  - All problems must be solved in Scheme
  - Make sure that you know how to use the Scheme interpreter by attending lab this week!

Announcements

- Homework 7 due Tuesday 11/5 @ 11:59pm.
- Project 1 composition revisions due Thursday 11/7 @ 11:59pm.
  - Instructions are posted on the course website (submit proj1revision)
- Homework 8 due Tuesday 11/12 @ 11:59pm.
  - All problems must be solved in Scheme
  - Make sure that you know how to use the Scheme interpreter by attending lab this week!
- An improved final exam score can partially make up for low midterm scores.

Announcements

- Homework 7 due Tuesday 11/5 @ 11:59pm.
- Project 1 composition revisions due Thursday 11/7 @ 11:59pm.
  - Instructions are posted on the course website (submit proj1revision)
- Homework 8 due Tuesday 11/12 @ 11:59pm.
  - All problems must be solved in Scheme
  - Make sure that you know how to use the Scheme interpreter by attending lab this week!
- An improved final exam score can partially make up for low midterm scores.
  - This policy will only affect students who might not otherwise pass the course.

Announcements

- Homework 7 due Tuesday 11/5 @ 11:59pm.
- Project 1 composition revisions due Thursday 11/7 @ 11:59pm.
  - Instructions are posted on the course website (submit proj1revision)
- Homework 8 due Tuesday 11/12 @ 11:59pm.
  - All problems must be solved in Scheme
  - Make sure that you know how to use the Scheme interpreter by attending lab this week!
- An improved final exam score can partially make up for low midterm scores.
  - This policy will only affect students who might not otherwise pass the course.
- Example for today: http://composingprograms.com/examples/scalc/scalc.html

Parsing

Parsing

Parsing

A Parser takes text and returns an expression.

Parsing

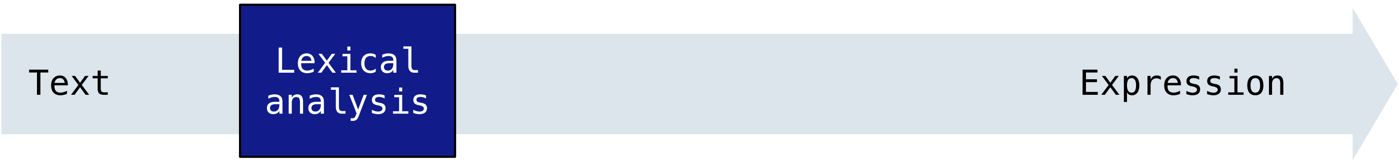
A Parser takes text and returns an expression.

Text

Expression

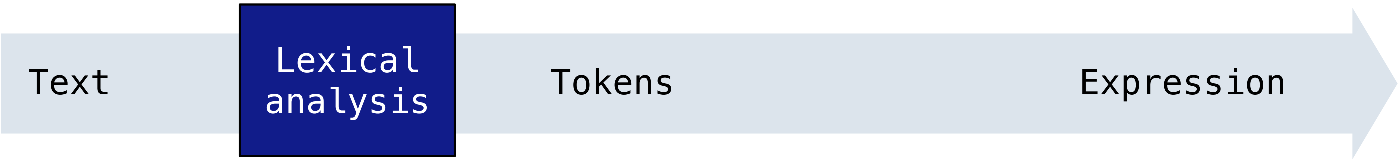
Parsing

A Parser takes text and returns an expression.



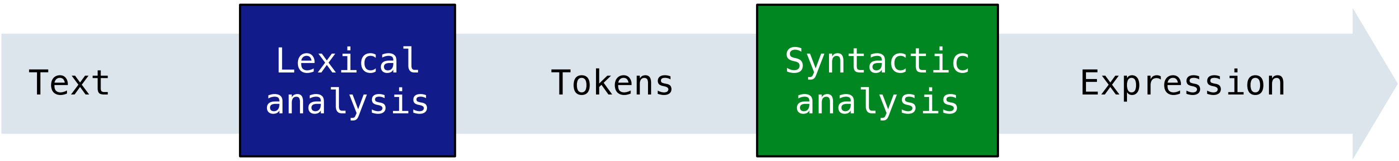
Parsing

A Parser takes text and returns an expression.



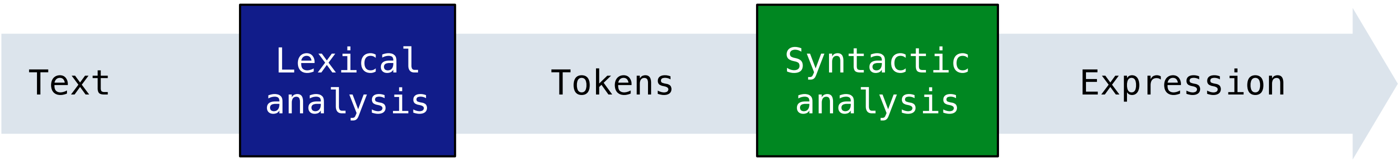
Parsing

A Parser takes text and returns an expression.



Parsing

A Parser takes text and returns an expression.



Text

Lexical analysis

Tokens

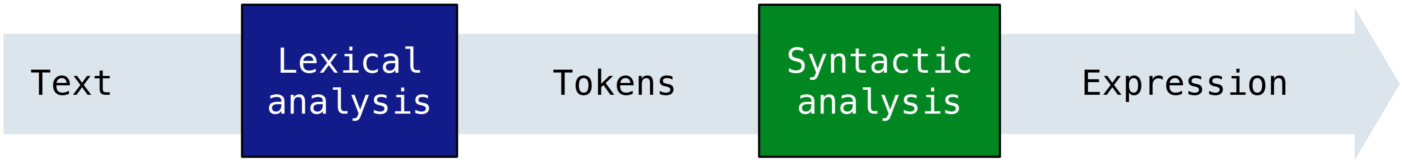
Syntactic analysis

Expression

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

Parsing

A Parser takes text and returns an expression.



Text

Lexical analysis

Tokens

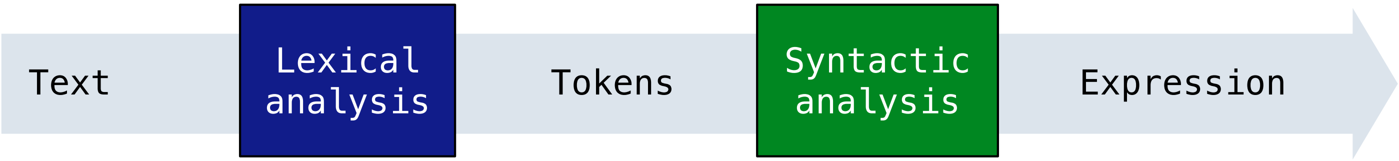
Syntactic analysis

Expression

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

Parsing

A Parser takes text and returns an expression.



Text

Lexical analysis

Tokens

Syntactic analysis

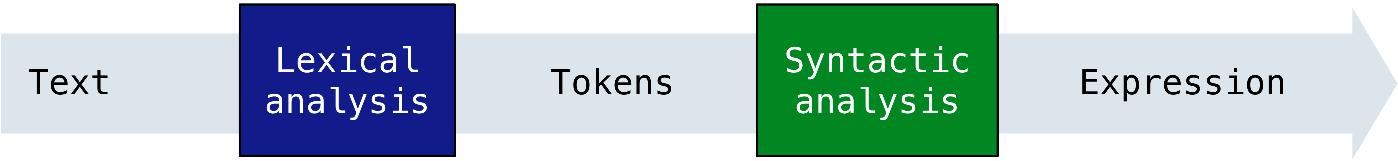
Expression

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

```
'(', '+', 1
```

Parsing

A Parser takes text and returns an expression.



Text

Lexical analysis

Tokens

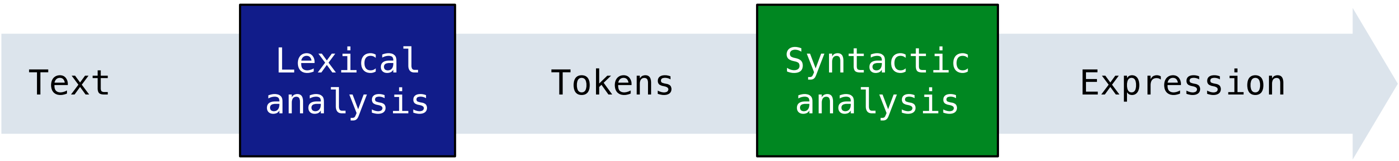
Syntactic analysis

Expression

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

Parsing

A Parser takes text and returns an expression.

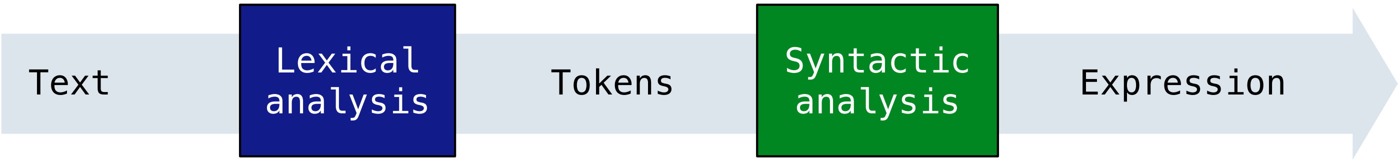


```
(+ 1
   (- 23)
   (* 4 5.6))
```

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

Parsing

A Parser takes text and returns an expression.

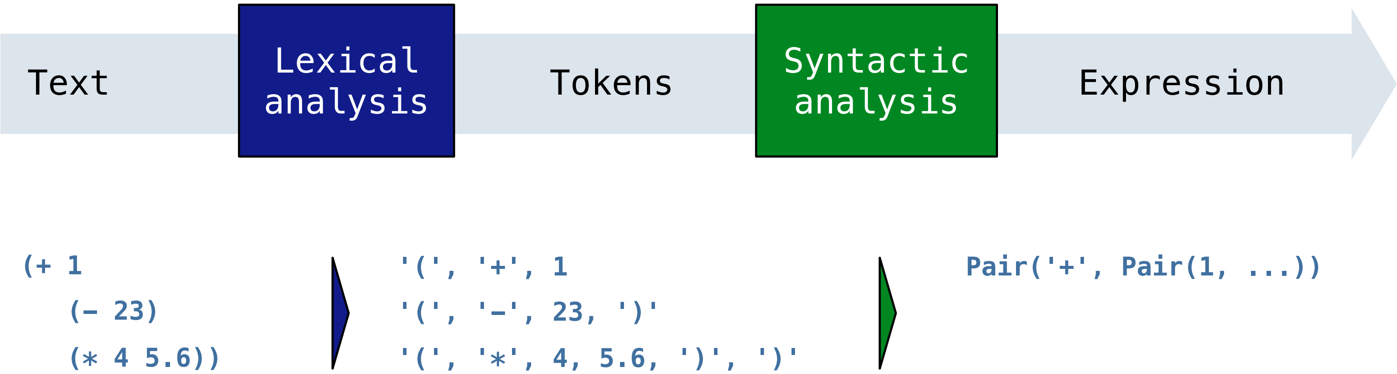


```
(+ 1
   (- 23)
   (* 4 5.6))
```

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

Parsing

A Parser takes text and returns an expression.



Text

Lexical analysis

Tokens

Syntactic analysis

Expression

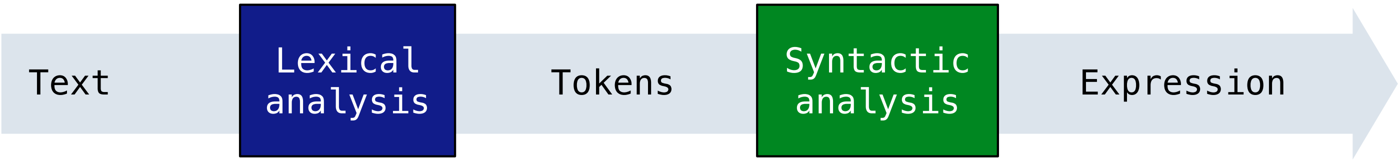
```
(+ 1
   (- 23)
   (* 4 5.6))
```

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

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

Parsing

A Parser takes text and returns an expression.



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

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

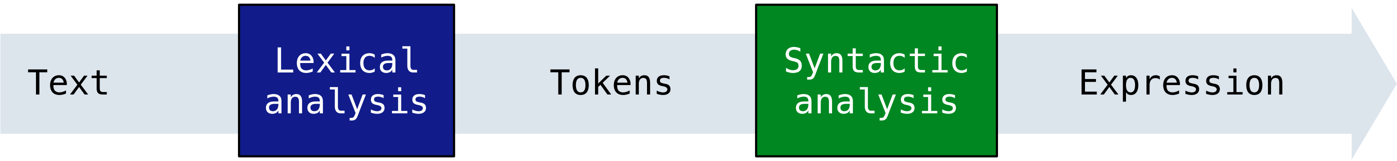
```
Pair('+', Pair(1, ...))
```

printed as

```
(+ 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))
```

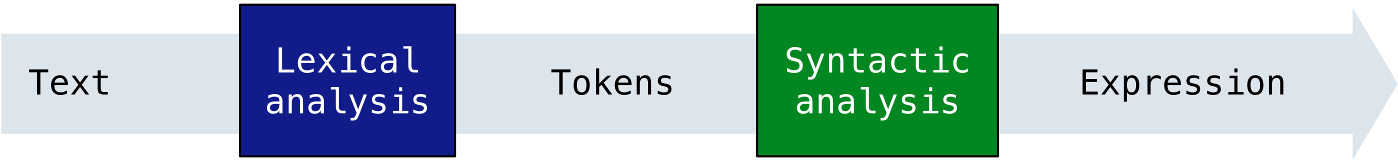
```
'(', '+', 1
'(', '-', 23, ')'
'(', '*', 4, 5.6, ')', ')'
```

```
Pair('+', Pair(1, ...))
        printed as
  (+ 1 (- 23) (* 4 5.6))
```

tokenize_line(line)
in scheme_tokens.py

Parsing

A Parser takes text and returns an expression.



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

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

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

printed as

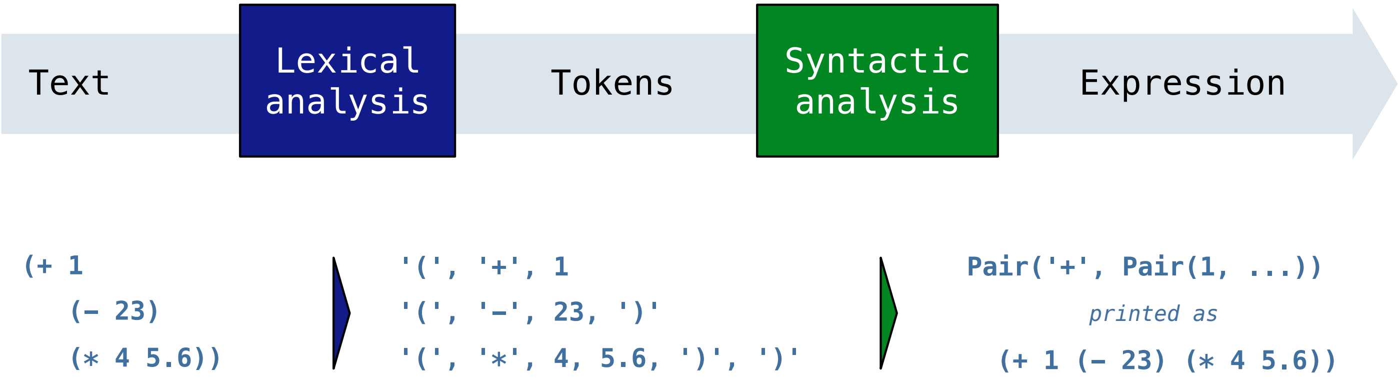
```
(+ 1 (- 23) (* 4 5.6))
```

tokenize_line(line)
in scheme_tokens.py

scheme_read(source)
scheme_reader.py

Parsing

A Parser takes text and returns an expression.



Text

Lexical analysis

Tokens

Syntactic analysis

Expression

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

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

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

printed as

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

tokenize_line(line)
in scheme_tokens.py

scheme_read(source)
scheme_reader.py

(Demo)

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.*

In Scheme, k is 1. The open-parenthesis starts a combination, the close-parenthesis ends a combination, and other tokens are primitive expressions.

Recursive Syntactic Analysis

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

In Scheme, k is 1. The open-parenthesis starts a combination, the close-parenthesis ends a combination, and other tokens are primitive expressions.

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*.

In Scheme, k is 1. The open-parenthesis starts a combination, the close-parenthesis ends a combination, and other tokens are primitive expressions.

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*.

In Scheme, k is 1. The open-parenthesis starts a combination, the close-parenthesis ends a combination, and other tokens are primitive expressions.

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*.

In Scheme, k is 1. The open-parenthesis starts a combination, the close-parenthesis ends a combination, and other tokens are primitive expressions.

Can English be parsed via predictive recursive descent?

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

ridden

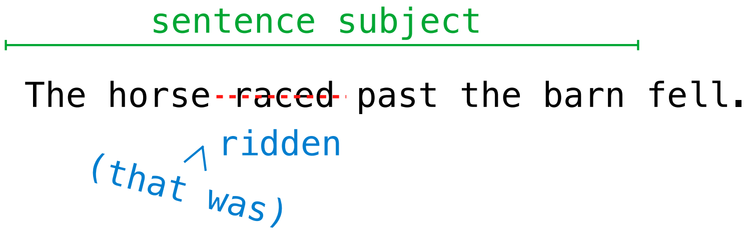
(that was)

Recursive Syntactic Analysis

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

In Scheme, k is 1. The open-parenthesis starts a combination, the close-parenthesis ends a combination, and other tokens are primitive expressions.

Can English be parsed via predictive recursive descent?



sentence subject

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

ridden

(that was)

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 are primitive expressions.

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 are primitive expressions.

Recursive call: scheme_read all 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 are primitive expressions.

Recursive call: scheme_read all 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 are primitive expressions.

Recursive call: scheme_read all 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 are primitive expressions.

Recursive call: scheme_read all 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 are primitive expressions.

Recursive call: scheme_read all 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 are primitive expressions.

Recursive call: scheme_read all 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 are primitive expressions.

Recursive call: scheme_read all 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 are primitive expressions.

Recursive call: scheme_read all 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 are primitive expressions.

Recursive call: scheme_read all 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 are primitive expressions.

Recursive call: scheme_read all sub-expressions and combine them.

(Demo)

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 3

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

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
```

Python 3 Byte Code

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

Programming Languages

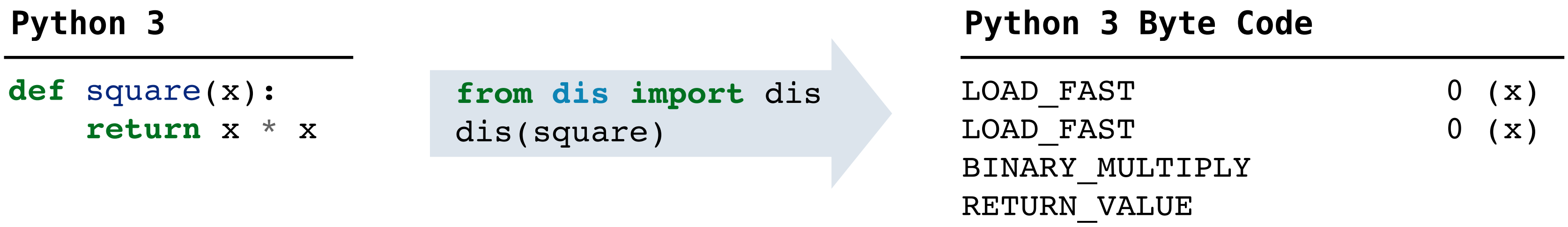
A computer typically executes programs written in many different programming languages.

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.

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.

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.

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.

```
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! *Homoiconic* means source code is data.

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

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))

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.

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.

Expression

```
(* 3
   (+ 4 5)
   (* 6 7 8))
```

Calculator Syntax

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

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

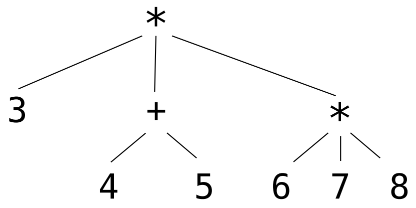
A call expression is a combination that begins with an operator (+, -, *, /) followed by 0 or more expressions: (+ 1 2 3), (/ 3 (+ 4 5))

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

Expression

```
(* 3
   (+ 4 5)
   (* 6 7 8))
```

Expression Tree



Calculator Syntax

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

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

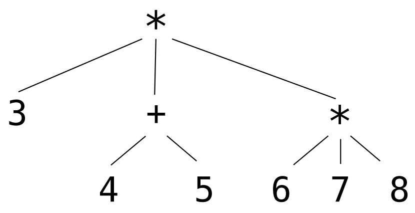
A call expression is a combination that begins with an operator (+, -, *, /) followed by 0 or more expressions: (+ 1 2 3), (/ 3 (+ 4 5))

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

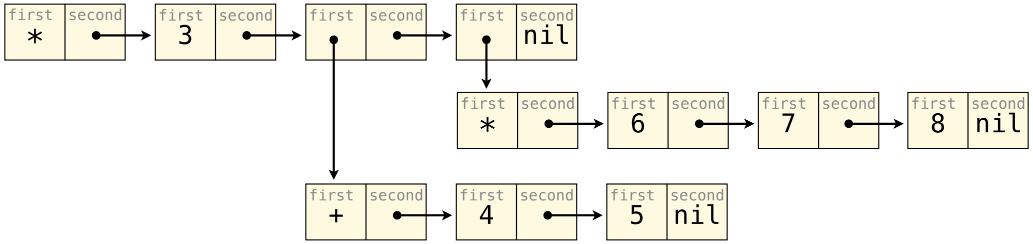
Expression

```
(* 3
   (+ 4 5)
   (* 6 7 8))
```

Expression Tree



Representation as Pairs



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

*****: 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

```
(* 3
   (+ 4 5)
   (* 6 7 8))
```

Calculator Semantics

The value of a calculator expression is defined recursively.

Primitive: A number evaluates to itself.

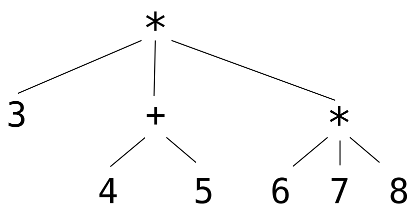
Call: A call expression evaluates to its argument values combined by an operator.

- **+**: 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

```
(* 3
   (+ 4 5)
   (* 6 7 8))
```

Expression Tree



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

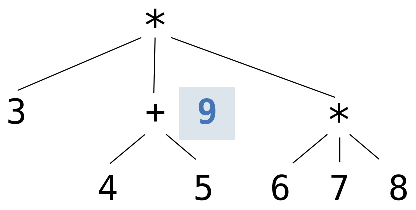
-: If one argument, negate it. If more than one, subtract the rest from the first.

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

Expression

```
(* 3
   (+ 4 5)
   (* 6 7 8))
```

Expression Tree



Calculator Semantics

The value of a calculator expression is defined recursively.

Primitive: A number evaluates to itself.

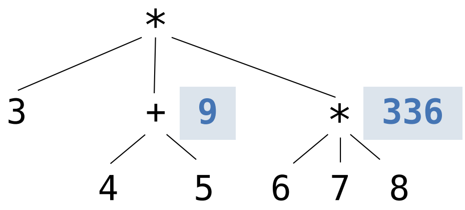
Call: A call expression evaluates to its argument values combined by an operator.

- **+**: 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

```
(* 3
   (+ 4 5)
   (* 6 7 8))
```

Expression Tree



Calculator Semantics

The value of a calculator expression is defined recursively.

Primitive: A number evaluates to itself.

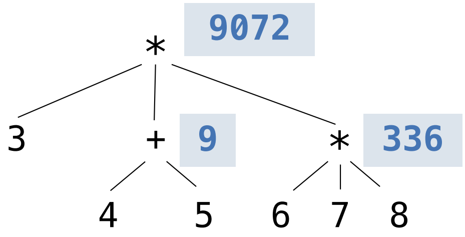
Call: A call expression evaluates to its argument values combined by an operator.

- **+**: 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

```
(* 3
   (+ 4 5)
   (* 6 7 8))
```

Expression Tree



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).

Implementation	**Language Semantics**

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

```
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

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

```
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

```
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

```
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

```
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).

Implementation

```
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

```
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
```

Recursive call returns a number for each operand

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

```
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
```

Recursive call returns a number for each operand

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

```
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
```

Recursive call returns a number for each operand

'+', '-', '*', '/'

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

```
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
```

Recursive call returns a number for each operand

'+', '-', '*', '/'

A Scheme list of numbers

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: +, -, *, /

Implementation

Language Semantics

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

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

Language Semantics

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

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

Language Semantics

+:

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

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

Language Semantics

+:

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

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

Language Semantics

+:

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

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

(Demo)

Language Semantics

+:
  Sum of the arguments
-:
  ...
...

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.

- Print a prompt.

Read-Eval-Print Loop

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

- Print a prompt.
- **Read** text input from the user.

Read-Eval-Print Loop

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

- Print a prompt.
- **Read** text input from the user.
- Parse the text input into an expression.

Read-Eval-Print Loop

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

- Print a prompt.
- **Read** text input from the user.
- Parse the text input into an expression.
- **Evaluate** the expression.

Read-Eval-Print Loop

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

- Print a prompt.
- **Read** text input from the user.
- Parse the text input into an expression.
- **Evaluate** the expression.
- If any errors occur, report those errors, otherwise

Read-Eval-Print Loop

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

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

Read-Eval-Print Loop

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

- Print a prompt.
- **Read** text input from the user.
- Parse the text input into an expression.
- **Evaluate** the expression.
- If any errors occur, report those errors, otherwise
- **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)