61A Lecture 26

Monday, October 31

Programming Languages

Programming Languages

Computers have software written in many different languages

Programming Languages

Computers have software written in many different languages

Machine languages: statements can be interpreted by hardware

Programming Languages

Computers have software written in many different languages

Machine languages: statements can be interpreted by hardware

- All data are represented as sequences of bits

Programming Languages

Computers have software written in many different languages

Machine languages: statements can be interpreted by hardware

- All data are represented as sequences of bits
- All statements are primitive instructions

Programming Languages

Computers have software written in many different languages

Machine languages: statements can be interpreted by hardware

- All data are represented as sequences of bits
- All statements are primitive instructions

High-level languages: hide concerns about those details

Programming Languages

Computers have software written in many different languages

Machine languages: statements can be interpreted by hardware

- All data are represented as sequences of bits
- All statements are primitive instructions

High-level languages: hide concerns about those details

- Primitive data types beyond just bits

Programming Languages

Computers have software written in many different languages

Machine languages: statements can be interpreted by hardware

- All data are represented as sequences of bits
- All statements are primitive instructions

High-level languages: hide concerns about those details

- Primitive data types beyond just bits
- Statements/expressions can be non-primitive (e.g., calls)

Programming Languages

Computers have software written in many different languages

Machine languages: statements can be interpreted by hardware

- All data are represented as sequences of bits
- All statements are primitive instructions

High-level languages: hide concerns about those details

- Primitive data types beyond just bits
- Statements/expressions can be non-primitive (e.g., calls)
- Evaluation process is defined in software, not hardware

Programming Languages

Computers have software written in many different languages

Machine languages: statements can be interpreted by hardware

- All data are represented as sequences of bits
- All statements are primitive instructions

High-level languages: hide concerns about those details

- Primitive data types beyond just bits
- Statements/expressions can be non-primitive (e.g., calls)
- Evaluation process is defined in software, not hardware

High-level languages are built on top of low-level languages

Programming Languages

Computers have software written in many different languages

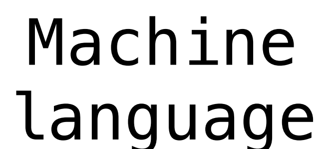
Machine languages: statements can be interpreted by hardware

- All data are represented as sequences of bits
- All statements are primitive instructions

High-level languages: hide concerns about those details

- Primitive data types beyond just bits
- Statements/expressions can be non-primitive (e.g., calls)
- Evaluation process is defined in software, not hardware

High-level languages are built on top of low-level languages



Programming Languages

Computers have software written in many different languages

Machine languages: statements can be interpreted by hardware

- All data are represented as sequences of bits
- All statements are primitive instructions

High-level languages: hide concerns about those details

- Primitive data types beyond just bits
- Statements/expressions can be non-primitive (e.g., calls)
- Evaluation process is defined in software, not hardware

High-level languages are built on top of low-level languages

Machine language	C

Programming Languages

Computers have software written in many different languages

Machine languages: statements can be interpreted by hardware

- All data are represented as sequences of bits
- All statements are primitive instructions

High-level languages: hide concerns about those details

- Primitive data types beyond just bits
- Statements/expressions can be non-primitive (e.g., calls)
- Evaluation process is defined in software, not hardware

High-level languages are built on top of low-level languages

Machine language	C	Python

Metalinguistic Abstraction

Metalinguistic Abstraction

Metalinguistic abstraction: Establishing new technical languages (such as programming languages)

Metalinguistic Abstraction

Metalinguistic abstraction: Establishing new technical languages (such as programming languages)

$$f(x) = x^2 - 2x + 1$$

Metalinguistic Abstraction

Metalinguistic abstraction: Establishing new technical languages (such as programming languages)

$$f(x) = x^2 - 2x + 1$$

$$\lambda f.(\lambda x.f(x\ x))(\lambda x.f(x\ x))$$

Metalinguistic Abstraction

Metalinguistic abstraction: Establishing new technical languages (such as programming languages)

$$f(x) = x^2 - 2x + 1$$

$$\lambda f.(\lambda x.f(x\ x))(\lambda x.f(x\ x))$$

In computer science, languages can be *implemented*:

Metalinguistic Abstraction

Metalinguistic abstraction: Establishing new technical languages (such as programming languages)

$$f(x) = x^2 - 2x + 1$$

$$\lambda f.(\lambda x.f(x\ x))(\lambda x.f(x\ x))$$

In computer science, languages can be *implemented*:

- An *interpreter* for a programming language is a function that, when applied to an expression of the language, performs the actions required to evaluate that expression

Metalinguistic Abstraction

Metalinguistic abstraction: Establishing new technical languages (such as programming languages)

$$f(x) = x^2 - 2x + 1$$

$$\lambda f.(\lambda x.f(x\ x))(\lambda x.f(x\ x))$$

In computer science, languages can be *implemented*:

- An *interpreter* for a programming language is a function that, when applied to an expression of the language, performs the actions required to evaluate that expression
- The *semantics* and *syntax* of a language must be specified precisely in order to allow for an interpreter

The Calculator Language

The Calculator Language

Prefix notation expression language for basic arithmetic

The Calculator Language

Prefix notation expression language for basic arithmetic

Python-like syntax, with more flexible built-in functions

The Calculator Language

Prefix notation expression language for basic arithmetic

Python-like syntax, with more flexible built-in functions

```
calc> add(1, 2, 3, 4)
10
```

The Calculator Language

Prefix notation expression language for basic arithmetic

Python-like syntax, with more flexible built-in functions

```
calc> add(1, 2, 3, 4)
10
calc> mul()
1
```

The Calculator Language

Prefix notation expression language for basic arithmetic

Python-like syntax, with more flexible built-in functions

```
calc> add(1, 2, 3, 4)
10
calc> mul()
1

calc> sub(100, mul(7, add(8, div(-12, -3))))
16.0
```

The Calculator Language

Prefix notation expression language for basic arithmetic

Python-like syntax, with more flexible built-in functions

```
calc> add(1, 2, 3, 4)
10
calc> mul()
1

calc> sub(100, mul(7, add(8, div(-12, -3))))
16.0
calc> -(100, *(7, +(8, /(-12, -3))))
16.0
```

The Calculator Language

Prefix notation expression language for basic arithmetic

Python-like syntax, with more flexible built-in functions

```
calc> add(1, 2, 3, 4)
10
calc> mul()
1

calc> sub(100, mul(7, add(8, div(-12, -3))))
16.0
calc> -(100, *(7, +(8, /(-12, -3))))
16.0
```

Demo

Syntax and Semantics of Calculator

Syntax and Semantics of Calculator

Expression types:

Syntax and Semantics of Calculator

Expression types:

- A **call expression** is an operator name followed by a comma-separated list of operand expressions, in parentheses

Syntax and Semantics of Calculator

Expression types:

- A **call expression** is an operator name followed by a comma-separated list of operand expressions, in parentheses
- A **primitive expression** is a number

Syntax and Semantics of Calculator

Expression types:

- A **call expression** is an operator name followed by a comma-separated list of operand expressions, in parentheses
- A **primitive expression** is a number

Operators:

Syntax and Semantics of Calculator

Expression types:

- A **call expression** is an operator name followed by a comma-separated list of operand expressions, in parentheses
- A **primitive expression** is a number

Operators:

- The {add,+} operator returns the sum of its arguments

Syntax and Semantics of Calculator

Expression types:

- A **call expression** is an operator name followed by a comma-separated list of operand expressions, in parentheses
- A **primitive expression** is a number

Operators:

- The {add,**+**} operator returns the sum of its arguments
- The {sub,**–**} operator returns either

Syntax and Semantics of Calculator

Expression types:

- A **call expression** is an operator name followed by a comma-separated list of operand expressions, in parentheses
- A **primitive expression** is a number

Operators:

- The {add,**+**} operator returns the sum of its arguments
- The {sub,**–**} operator returns either
  - the additive inverse of a single argument, or

Syntax and Semantics of Calculator

Expression types:

- A **call expression** is an operator name followed by a comma-separated list of operand expressions, in parentheses
- A **primitive expression** is a number

Operators:

- The {add,+} operator returns the sum of its arguments
- The {sub,–} operator returns either
  - the additive inverse of a single argument, or
  - the sum of subsequent arguments subtracted from the first

Syntax and Semantics of Calculator

Expression types:

- A **call expression** is an operator name followed by a comma-separated list of operand expressions, in parentheses
- A **primitive expression** is a number

Operators:

- The {add,+} operator returns the sum of its arguments
- The {sub,-} operator returns either
  - the additive inverse of a single argument, or
  - the sum of subsequent arguments subtracted from the first
- The {mul,*} operator returns the product of its arguments

Syntax and Semantics of Calculator

Expression types:

- A **call expression** is an operator name followed by a comma-separated list of operand expressions, in parentheses
- A **primitive expression** is a number

Operators:

- The {add,+} operator returns the sum of its arguments
- The {sub,-} operator returns either
  - the additive inverse of a single argument, or
  - the sum of subsequent arguments subtracted from the first
- The {mul,*} operator returns the product of its arguments
- The {div,/} operator returns the real-valued quotient of a dividend and divisor (i.e., a numerator and denominator)

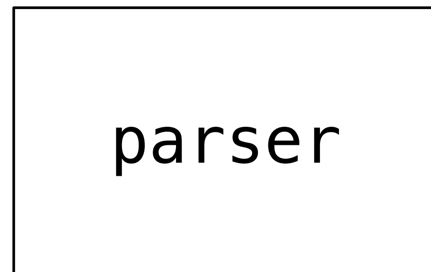
Expression Trees

Expression Trees

A basic interpreter has two parts: a *parser* and an *evaluator*

Expression Trees

A basic interpreter has two parts: a *parser* and an *evaluator*



Expression Trees

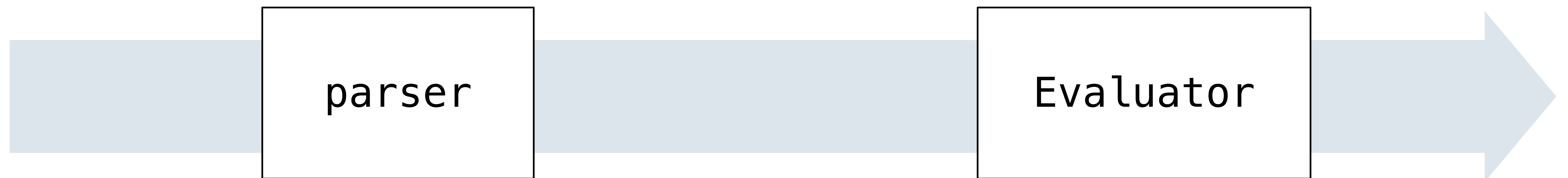
A basic interpreter has two parts: a *parser* and an *evaluator*

parser

Evaluator

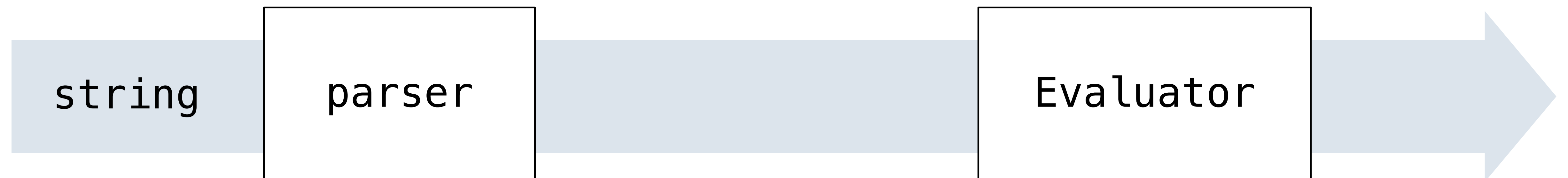
Expression Trees

A basic interpreter has two parts: a *parser* and an *evaluator*



Expression Trees

A basic interpreter has two parts: a *parser* and an *evaluator*



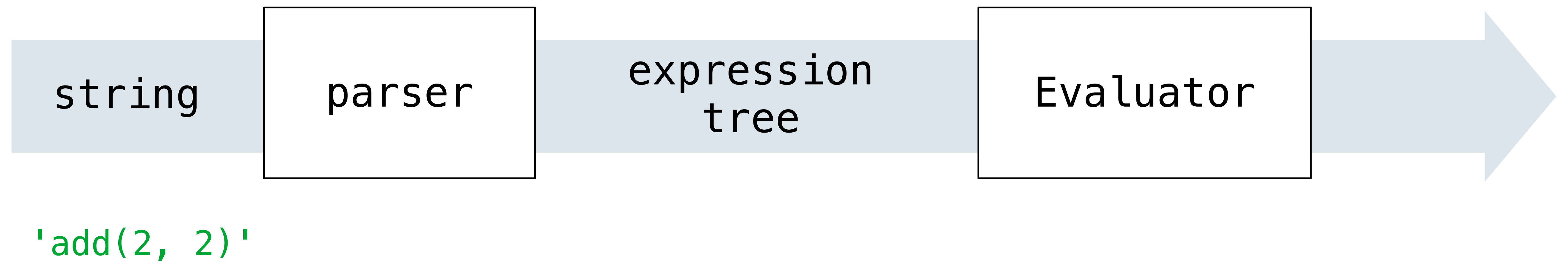
Expression Trees

A basic interpreter has two parts: a *parser* and an *evaluator*

'add(2, 2)'

Expression Trees

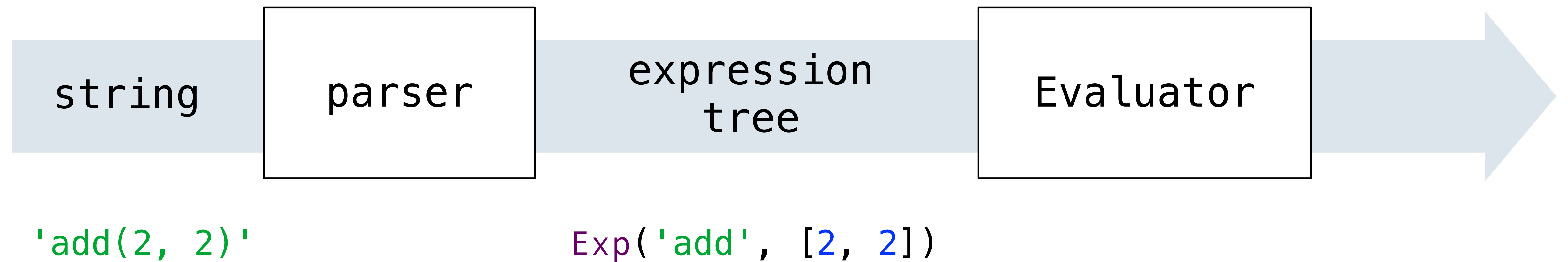
A basic interpreter has two parts: a *parser* and an *evaluator*



'add(2, 2)'

Expression Trees

A basic interpreter has two parts: a *parser* and an *evaluator*

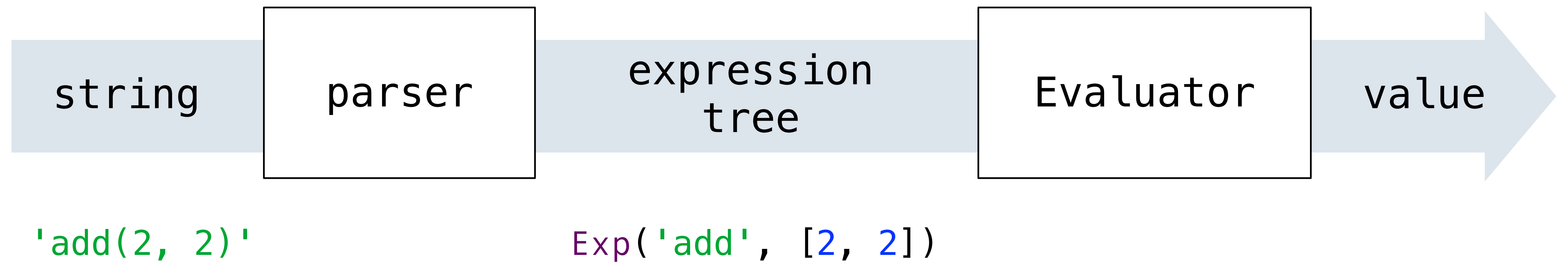


'add(2, 2)'

Exp('add', [2, 2])

Expression Trees

A basic interpreter has two parts: a *parser* and an *evaluator*

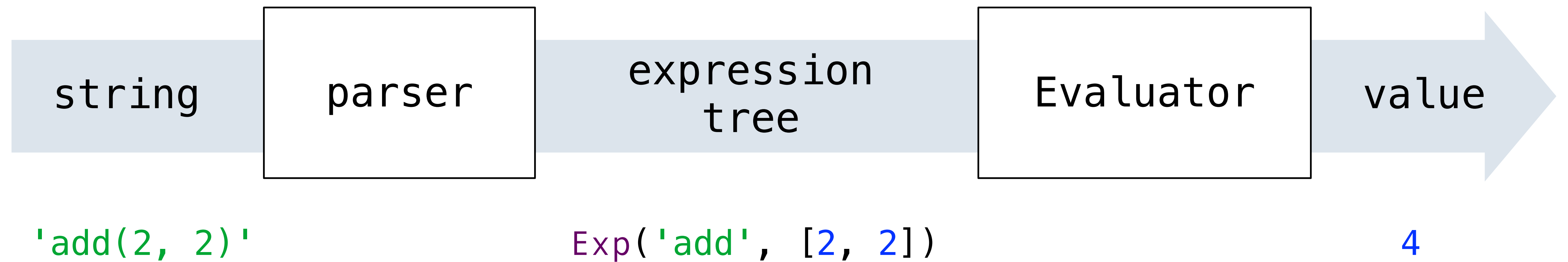


'add(2, 2)'

Exp('add', [2, 2])

Expression Trees

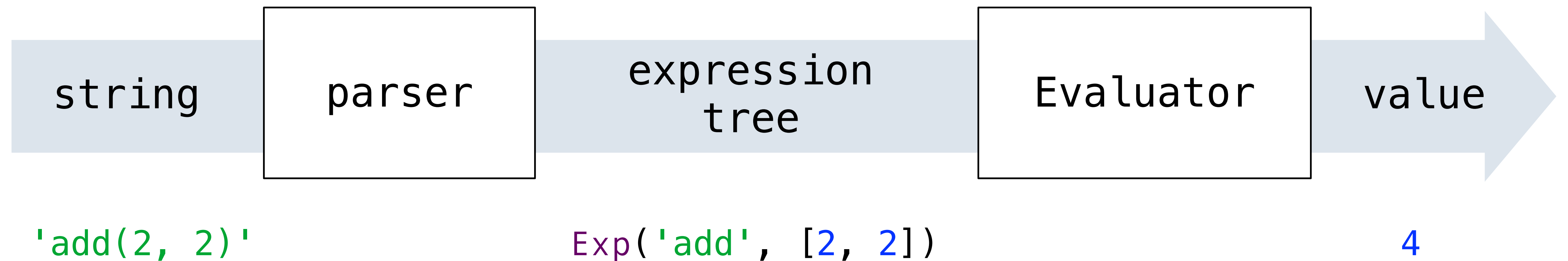
A basic interpreter has two parts: a *parser* and an *evaluator*



'add(2, 2)' Exp('add', [2, 2]) 4

Expression Trees

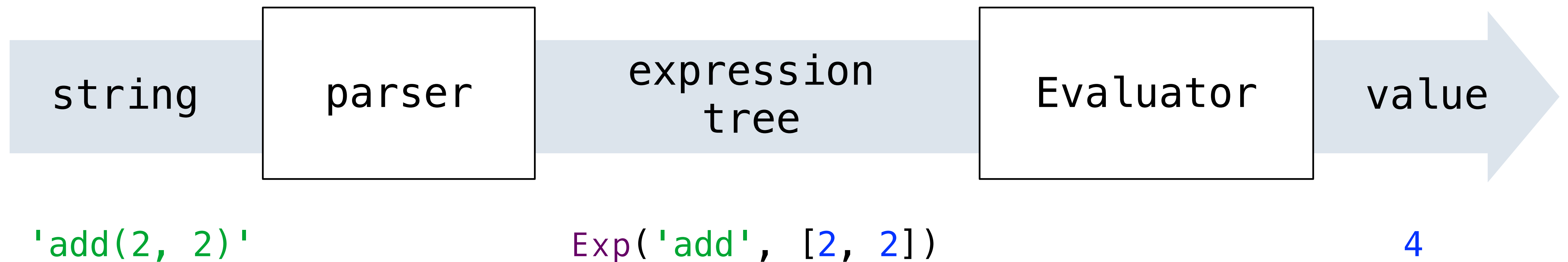
A basic interpreter has two parts: a *parser* and an *evaluator*



An *expression tree* is a (hierarchical) data structure that represents a (nested) expression

Expression Trees

A basic interpreter has two parts: a *parser* and an *evaluator*



'add(2, 2)'   Exp('add', [2, 2])   4

An *expression tree* is a (hierarchical) data structure that represents a (nested) expression

```
class Exp(object):
    """A call expression in Calculator."""
    def __init__(self, operator, operands):
        self.operator = operator
        self.operands = operands
```

Creating Expression Trees Directly

Creating Expression Trees Directly

We can construct expression trees in Python directly

Creating Expression Trees Directly

We can construct expression trees in Python directly

The __str__ method of Exp returns a Calculator call expression

Creating Expression Trees Directly

We can construct expression trees in Python directly

The __str__ method of Exp returns a Calculator call expression

```
>>> Exp('add', [1, 2])
Exp('add', [1, 2])
```

Creating Expression Trees Directly

We can construct expression trees in Python directly

The __str__ method of Exp returns a Calculator call expression

```
>>> Exp('add', [1, 2])
Exp('add', [1, 2])

>>> str(Exp('add', [1, 2]))
'add(1, 2)'
```

Creating Expression Trees Directly

We can construct expression trees in Python directly

The __str__ method of Exp returns a Calculator call expression

```
>>> Exp('add', [1, 2])
Exp('add', [1, 2])

>>> str(Exp('add', [1, 2]))
'add(1, 2)'

>>> Exp('add', [1, Exp('mul', [2, 3, 4])])
Exp('add', [1, Exp('mul', [2, 3, 4])])
```

Creating Expression Trees Directly

We can construct expression trees in Python directly

The __str__ method of Exp returns a Calculator call expression

```
>>> Exp('add', [1, 2])
Exp('add', [1, 2])

>>> str(Exp('add', [1, 2]))
'add(1, 2)'

>>> Exp('add', [1, Exp('mul', [2, 3, 4])])
Exp('add', [1, Exp('mul', [2, 3, 4])])

>>> str(Exp('add', [1, Exp('mul', [2, 3, 4])]))
'add(1, mul(2, 3, 4))'
```

Evaluation

Evaluation

Evaluation discovers the form of an expression and then executes a corresponding evaluation rule.

Evaluation

Evaluation discovers the form of an expression and then executes a corresponding evaluation rule.

- Primitive expressions (literals) are evaluated directly

Evaluation

Evaluation discovers the form of an expression and then executes a corresponding evaluation rule.

- Primitive expressions (literals) are evaluated directly
- Call expressions are evaluated recursively

Evaluation

Evaluation discovers the form of an expression and then executes a corresponding evaluation rule.

- Primitive expressions (literals) are evaluated directly
- Call expressions are evaluated recursively
  - Evaluate each operand expression

Evaluation

Evaluation discovers the form of an expression and then executes a corresponding evaluation rule.

- Primitive expressions (literals) are evaluated directly
- Call expressions are evaluated recursively
  - Evaluate each operand expression
  - Collect their values as a list of arguments

Evaluation

Evaluation discovers the form of an expression and then executes a corresponding evaluation rule.

- Primitive expressions (literals) are evaluated directly
- Call expressions are evaluated recursively
  - Evaluate each operand expression
  - Collect their values as a list of arguments
  - *Apply* the named operator to the argument list

Evaluation

Evaluation discovers the form of an expression and then executes a corresponding evaluation rule.

- Primitive expressions (literals) are evaluated directly
- Call expressions are evaluated recursively
  - Evaluate each operand expression
  - Collect their values as a list of arguments
  - *Apply* the named operator to the argument list

```
def calc_eval(exp):
    """Evaluate a Calculator expression."""
```

Evaluation

Evaluation discovers the form of an expression and then executes a corresponding evaluation rule.

- Primitive expressions (literals) are evaluated directly
- Call expressions are evaluated recursively
  - Evaluate each operand expression
  - Collect their values as a list of arguments
  - *Apply* the named operator to the argument list

```
def calc_eval(exp):
    """Evaluate a Calculator expression."""
    if type(exp) in (int, float):
        return exp
```

Evaluation

Evaluation discovers the form of an expression and then executes a corresponding evaluation rule.

- Primitive expressions (literals) are evaluated directly
- Call expressions are evaluated recursively
  - Evaluate each operand expression
  - Collect their values as a list of arguments
  - *Apply* the named operator to the argument list

```
def calc_eval(exp):
    """Evaluate a Calculator expression."""
    if type(exp) in (int, float):
        return exp
```

Numbers are self-evaluating

Evaluation

Evaluation discovers the form of an expression and then executes a corresponding evaluation rule.

- Primitive expressions (literals) are evaluated directly
- Call expressions are evaluated recursively
  - Evaluate each operand expression
  - Collect their values as a list of arguments
  - *Apply* the named operator to the argument list

```
def calc_eval(exp):
    """Evaluate a Calculator expression."""
    if type(exp) in (int, float):
        return exp
    elif type(exp) == Exp:
```

Evaluation

Evaluation discovers the form of an expression and then executes a corresponding evaluation rule.

- Primitive expressions (literals) are evaluated directly
- Call expressions are evaluated recursively
  - Evaluate each operand expression
  - Collect their values as a list of arguments
  - *Apply* the named operator to the argument list

```
def calc_eval(exp):
    """Evaluate a Calculator expression."""
    if type(exp) in (int, float):
        return exp
    elif type(exp) == Exp:
        arguments = list(map(calc_eval, exp.operands))
```

Numbers are self-evaluating

Evaluation

Evaluation discovers the form of an expression and then executes a corresponding evaluation rule.

- Primitive expressions (literals) are evaluated directly
- Call expressions are evaluated recursively
  - Evaluate each operand expression
  - Collect their values as a list of arguments
  - *Apply* the named operator to the argument list

```
def calc_eval(exp):
    """Evaluate a Calculator expression."""
    if type(exp) in (int, float):
        return exp
    elif type(exp) == Exp:
        arguments = list(map(calc_eval, exp.operands))
        return calc_apply(exp.operator, arguments)
```

Numbers are self-evaluating

Applying Operators

Applying Operators

`Calculator has a fixed set of operators that we can enumerate`

Applying Operators

Calculator has a fixed set of operators that we can enumerate

```
def calc_apply(operator, args):
    """Apply the named operator to a list of args."""
```

Applying Operators

Calculator has a fixed set of operators that we can enumerate

```
def calc_apply(operator, args):
    """Apply the named operator to a list of args."""
    if operator in ('add', '+'):
        return sum(args)
```

Applying Operators

Calculator has a fixed set of operators that we can enumerate

```
def calc_apply(operator, args):
    """Apply the named operator to a list of args."""
    if operator in ('add', '+'):
        return sum(args)
```

Applying Operators

Calculator has a fixed set of operators that we can enumerate

```
def calc_apply(operator, args):
    """Apply the named operator to a list of args."""
    if operator in ('add', '+'):
        return sum(args)
    if operator in ('sub', '-'):
        if len(args) == 1:
            return -args[0]
        return sum(args[:1] + [-arg for arg in args[1:]])
    ...
```

Applying Operators

Calculator has a fixed set of operators that we can enumerate

```
def calc_apply(operator, args):
    """Apply the named operator to a list of args."""
    if operator in ('add', '+'):
        return sum(args)
    if operator in ('sub', '-'):
        if len(args) == 1:
            return -args[0]
        return sum(args[:1] + [-arg for arg in args[1:]])
    ...
```

Dispatch on operator name

Implement operator logic in Python

Applying Operators

Calculator has a fixed set of operators that we can enumerate

```
def calc_apply(operator, args):
    """Apply the named operator to a list of args."""
    if operator in ('add', '+'):
        return sum(args)
    if operator in ('sub', '-'):
        if len(args) == 1:
            return -args[0]
        return sum(args[:1] + [-arg for arg in args[1:]])
    ...
```

Dispatch on operator name

Implement operator logic in Python

Demo

Read-Eval-Print Loop

Read-Eval-Print Loop

The user interface to many programming languages is an interactive loop, which

Read-Eval-Print Loop

The user interface to many programming languages is an interactive loop, which

- Reads an expression from the user

Read-Eval-Print Loop

The user interface to many programming languages is an interactive loop, which

- Reads an expression from the user
- Parses the input to build an expression tree

Read-Eval-Print Loop

The user interface to many programming languages is an interactive loop, which

- Reads an expression from the user
- Parses the input to build an expression tree
- Evaluates the expression tree

Read-Eval-Print Loop

The user interface to many programming languages is an interactive loop, which

- Reads an expression from the user
- Parses the input to build an expression tree
- Evaluates the expression tree
- Prints the resulting value of the expression

Read-Eval-Print Loop

The user interface to many programming languages is an interactive loop, which

- Reads an expression from the user
- Parses the input to build an expression tree
- Evaluates the expression tree
- Prints the resulting value of the expression

```
def read_eval_print_loop():
    """Run a read-eval-print loop for calculator."""
    while True:
        expression_tree = calc_parse(input('calc> '))
        print(calc_eval(expression_tree))
```

Read-Eval-Print Loop

The user interface to many programming languages is an interactive loop, which

- Reads an expression from the user
- Parses the input to build an expression tree
- Evaluates the expression tree
- Prints the resulting value of the expression

```
def read_eval_print_loop():
    """Run a read-eval-print loop for calculator."""
    while True:
        expression_tree = calc_parse(input('calc> '))
        print(calc_eval(expression_tree))
```

Raising Application Errors

Raising Application Errors

The sub and div operators have restrictions on argument number

Raising Application Errors

The sub and div operators have restrictions on argument number

Raising exceptions in *apply* can identify such issues

Raising Application Errors

The sub and div operators have restrictions on argument number

Raising exceptions in *apply* can identify such issues

```
def calc_apply(operator, args):
```

Raising Application Errors

The sub and div operators have restrictions on argument number

Raising exceptions in *apply* can identify such issues

```
def calc_apply(operator, args):
    """Apply the named operator to a list of args."""
```

Raising Application Errors

The sub and div operators have restrictions on argument number

Raising exceptions in *apply* can identify such issues

```
def calc_apply(operator, args):
    """Apply the named operator to a list of args."""
    ...
```

Raising Application Errors

The sub and div operators have restrictions on argument number

Raising exceptions in *apply* can identify such issues

```
def calc_apply(operator, args):
    """Apply the named operator to a list of args."""
    ...
    if operator in ('sub', '-'):
```

Raising Application Errors

The sub and div operators have restrictions on argument number

Raising exceptions in *apply* can identify such issues

```
def calc_apply(operator, args):
    """Apply the named operator to a list of args."""
    ...
    if operator in ('sub', '-'):
        if len(args) == 0:
```

Raising Application Errors

The sub and div operators have restrictions on argument number

Raising exceptions in *apply* can identify such issues

```
def calc_apply(operator, args):
    """Apply the named operator to a list of args."""
    ...
    if operator in ('sub', '-'):
        if len(args) == 0:
            raise TypeError(operator + ' requires at least 1 argument')
```

Raising Application Errors

The sub and div operators have restrictions on argument number

Raising exceptions in *apply* can identify such issues

```
def calc_apply(operator, args):
    """Apply the named operator to a list of args."""
    ...
    if operator in ('sub', '-'):
        if len(args) == 0:
            raise TypeError(operator + ' requires at least 1 argument')
        ...
```

Raising Application Errors

The sub and div operators have restrictions on argument number

Raising exceptions in *apply* can identify such issues

```
def calc_apply(operator, args):
    """Apply the named operator to a list of args."""
    ...
    if operator in ('sub', '-'):
        if len(args) == 0:
            raise TypeError(operator + ' requires at least 1 argument')
        ...
    ...
```

Raising Application Errors

The sub and div operators have restrictions on argument number

Raising exceptions in *apply* can identify such issues

```
def calc_apply(operator, args):
    """Apply the named operator to a list of args."""
    ...
    if operator in ('sub', '-'):
        if len(args) == 0:
            raise TypeError(operator + ' requires at least 1 argument')
        ...
    ...
    if operator in ('div', '/'):
```

Raising Application Errors

The sub and div operators have restrictions on argument number

Raising exceptions in *apply* can identify such issues

```
def calc_apply(operator, args):
    """Apply the named operator to a list of args."""
    ...
    if operator in ('sub', '-'):
        if len(args) == 0:
            raise TypeError(operator + ' requires at least 1 argument')
        ...
    ...
    if operator in ('div', '/'):
        if len(args) != 2:
```

Raising Application Errors

The sub and div operators have restrictions on argument number

Raising exceptions in *apply* can identify such issues

```
def calc_apply(operator, args):
    """Apply the named operator to a list of args."""
    ...
    if operator in ('sub', '-'):
        if len(args) == 0:
            raise TypeError(operator + ' requires at least 1 argument')
        ...
    ...
    if operator in ('div', '/'):
        if len(args) != 2:
            raise TypeError(operator + ' requires exactly 2 arguments')
```

Raising Application Errors

The sub and div operators have restrictions on argument number

Raising exceptions in *apply* can identify such issues

```
def calc_apply(operator, args):
    """Apply the named operator to a list of args."""
    ...
    if operator in ('sub', '-'):
        if len(args) == 0:
            raise TypeError(operator + ' requires at least 1 argument')
        ...
    ...
    if operator in ('div', '/'):
        if len(args) != 2:
            raise TypeError(operator + ' requires exactly 2 arguments')
        ...
```

Handling Errors

Handling Errors

The REPL handles errors by printing informative messages for the user, rather than crashing.

Handling Errors

The REPL handles errors by printing informative messages for the user, rather than crashing.

```
def read_eval_print_loop():
    """Run a read-eval-print loop for calculator."""
```

Handling Errors

The REPL handles errors by printing informative messages for the user, rather than crashing.

```
def read_eval_print_loop():
    """Run a read-eval-print loop for calculator."""
    while True:
```

Handling Errors

The REPL handles errors by printing informative messages for the user, rather than crashing.

```
def read_eval_print_loop():
    """Run a read-eval-print loop for calculator."""
    while True:
        try:
            expression_tree = calc_parse(input('calc> '))
            print(calc_eval(expression_tree))
```

Handling Errors

The REPL handles errors by printing informative messages for the user, rather than crashing.

```
def read_eval_print_loop():
    """Run a read-eval-print loop for calculator."""
    while True:
        try:
            expression_tree = calc_parse(input('calc> '))
            print(calc_eval(expression_tree))
        except (SyntaxError, TypeError, ZeroDivisionError) as err:
            print(type(err).__name__ + ':', err)
```

Handling Errors

The REPL handles errors by printing informative messages for the user, rather than crashing.

```
def read_eval_print_loop():
    """Run a read-eval-print loop for calculator."""
    while True:
        try:
            expression_tree = calc_parse(input('calc> '))
            print(calc_eval(expression_tree))
        except (SyntaxError, TypeError, ZeroDivisionError) as err:
            print(type(err).__name__ + ':', err)
        except (KeyboardInterrupt, EOFError):  # <Control>-D, etc.
            print('Calculation completed.')
            return
```

Handling Errors

The REPL handles errors by printing informative messages for the user, rather than crashing.

```
def read_eval_print_loop():
    """Run a read-eval-print loop for calculator."""
    while True:
        try:
            expression_tree = calc_parse(input('calc> '))
            print(calc_eval(expression_tree))
        except (SyntaxError, TypeError, ZeroDivisionError) as err:
            print(type(err).__name__ + ':', err)
        except (KeyboardInterrupt, EOFError):  # <Control>-D, etc.
            print('Calculation completed.')
            return
```

A well-designed REPL should not crash on any input!

Handling Errors

The REPL handles errors by printing informative messages for the user, rather than crashing.

```
def read_eval_print_loop():
    """Run a read-eval-print loop for calculator."""
    while True:
        try:
            expression_tree = calc_parse(input('calc> '))
            print(calc_eval(expression_tree))
        except (SyntaxError, TypeError, ZeroDivisionError) as err:
            print(type(err).__name__ + ':', err)
        except (KeyboardInterrupt, EOFError):  # <Control>-D, etc.
            print('Calculation completed.')
            return
```

A well-designed REPL should not crash on any input!

Demo