61A Lecture 27

Wednesday, 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

Machine language

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 build an interpreter.

The Scheme-Syntax Calculator Language

The Scheme-Syntax Calculator Language

A subset of Scheme that includes:

- Number primitives
- Built-in arithmetic operators: +, -, *, /
- Call expressions

The Scheme-Syntax Calculator Language

A subset of Scheme that includes:

- Number primitives
- Built-in arithmetic operators: +, −, *, /
- Call expressions

```
> (+ (* 3 5) (- 10 6))
19

> (+ (* 3
        (+ (* 2 4)
           (+ 3 5)))
     (+ (- 10 7)
        6))
57
```

Syntax and Semantics of Calculator

Syntax and Semantics of Calculator

Expression types:

Syntax and Semantics of Calculator

Expression types:

- A **call expression** is a Scheme list

Syntax and Semantics of Calculator

Expression types:

- A **call expression** is a Scheme list
- A **primitive expression** is an operator symbol or number

Syntax and Semantics of Calculator

Expression types:

- A **call expression** is a Scheme list
- A **primitive expression** is an operator symbol or number

Operators:

Syntax and Semantics of Calculator

Expression types:

- A **call expression** is a Scheme list
- A **primitive expression** is an operator symbol or number

Operators:

- The + operator returns the sum of its arguments

Syntax and Semantics of Calculator

Expression types:

- A **call expression** is a Scheme list
- A **primitive expression** is an operator symbol or number

Operators:

- The + operator returns the sum of its arguments
- The – operator returns either

Syntax and Semantics of Calculator

Expression types:

- A **call expression** is a Scheme list
- A **primitive expression** is an operator symbol or number

Operators:

- The + operator returns the sum of its arguments
- The – operator returns either
  - the additive inverse of a single argument, or

Syntax and Semantics of Calculator

Expression types:

- A **call expression** is a Scheme list
- A **primitive expression** is an operator symbol or number

Operators:

- The + operator returns the sum of its arguments
- The – 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 a Scheme list
- A **primitive expression** is an operator symbol or number

Operators:

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

Syntax and Semantics of Calculator

Expression types:

- A **call expression** is a Scheme list
- A **primitive expression** is an operator symbol or number

Operators:

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

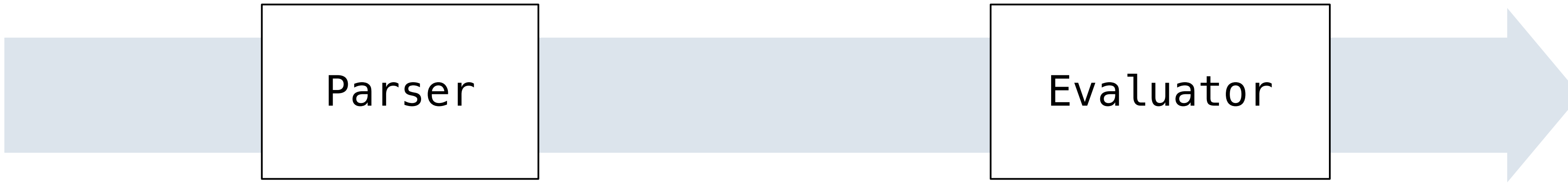
Parser

Expression Trees

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

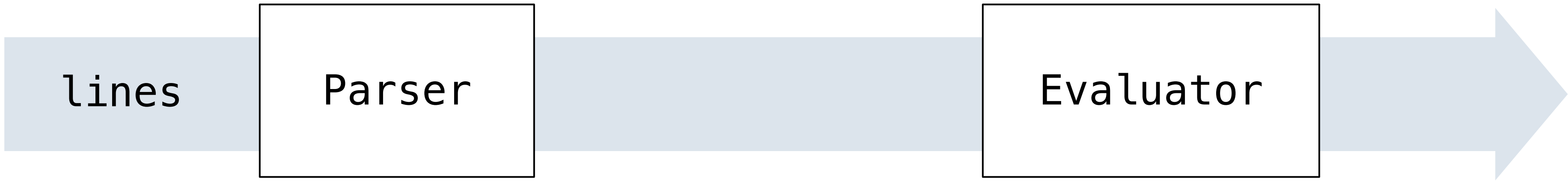
Expression Trees

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



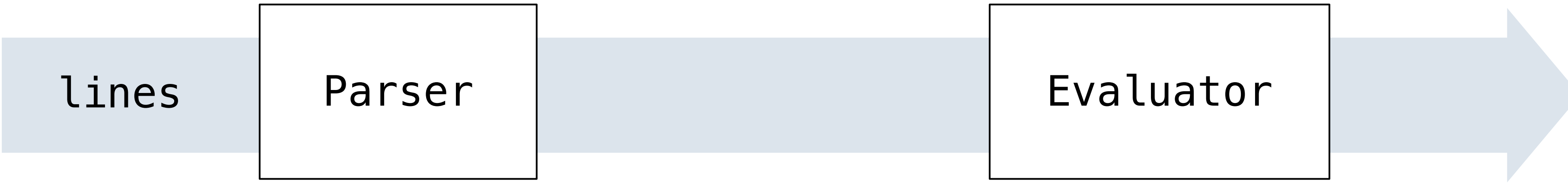
Expression Trees

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



Expression Trees

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



'(+ 2 2)'

Expression Trees

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

`'(+ 2 2)'`

Expression Trees

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


```
'(+ 2 2)'                Pair('+', Pair(2, Pair(2, nil)))
```

Expression Trees

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

'(+ 2 2)'

Pair('+', Pair(2, Pair(2, nil)))

Expression Trees

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

'(+ 2 2)' **Pair('+', Pair(2, Pair(2, nil)))** **4**

Expression Trees

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


```
'(+ 2 2)'            Pair('+', Pair(2, Pair(2, nil)))            4
```

```
'(* (+ 1'
'      (- 23)'
'      (* 4 5.6))'
'   10)'
```

Expression Trees

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


```
'(+ 2 2)'                 Pair('+', Pair(2, Pair(2, nil)))          4
```

```
'(* (+ 1'
'      (- 23)'
'      (* 4 5.6))'
'   10)'
```

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

printed as

```
(* (+ 1 (- 23) (* 4 5.6)) 10)
```

Expression Trees

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


```
'(+ 2 2)'                Pair('+', Pair(2, Pair(2, nil)))              4

'(* (+ 1'
'      (- 23)'           Pair('*', Pair(Pair('+', ...)))               4
'      (* 4 5.6))'                  printed as
'   10)'                 (* (+ 1 (- 23) (* 4 5.6)) 10)
```

Expression Trees

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


```
'(+ 2 2)'                    Pair('+', Pair(2, Pair(2, nil)))                    4
```

```
'(* (+ 1'
'      (- 23)'
'      (* 4 5.6))'
'   10)'
```

```
Pair('*', Pair(Pair('+', ...)))                    4
```

printed as

```
(* (+ 1 (- 23) (* 4 5.6)) 10)
```

Lines forming a Scheme expression

Expression Trees

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

lines → **Parser** → expression → **Evaluator** → value

lines	expression	value
`'(+ 2 2)'`	`Pair('+', Pair(2, Pair(2, nil)))`	`4`

`'(* (+ 1'`
`'      (- 23)'`
`'      (* 4 5.6))'`
`'   10)'`
	`Pair('*', Pair(Pair('+', ...)))`
	printed as
	`(* (+ 1 (- 23) (* 4 5.6)) 10)`
		`4`

Lines forming a Scheme expression	A number or a Pair with an operator as its first element	

Expression Trees

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

lines	expression	value
`'(+ 2 2)'`	`Pair('+', Pair(2, Pair(2, nil)))`	`4`

`'(* (+ 1'`
`'      (- 23)'`
`'      (* 4 5.6))'`
`'   10)'`
	`Pair('*', Pair(Pair('+', ...)))`
	printed as
	`(* (+ 1 (- 23) (* 4 5.6)) 10)`
		`4`

Lines forming a Scheme expression	**A number or a Pair with an operator as its first element**	**A number**

Expression Trees

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

lines	expression	value
`'(+ 2 2)'`	`Pair('+', Pair(2, Pair(2, nil)))`	4

`'(* (+ 1'`
`'      (- 23)'`
`'      (* 4 5.6))'`
`'   10)'`
	`Pair('*', Pair(Pair('+', ...)))`
	printed as
	`(* (+ 1 (- 23) (* 4 5.6)) 10)`
		4

Lines forming a Scheme expression	A number or a Pair with an operator as its first element	A number

Expression Trees

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

scheme_reader.py | **scalc.py**

lines → Parser → expression → Evaluator → value

lines	expression	value
`'(+ 2 2)'`	`Pair('+', Pair(2, Pair(2, nil)))`	**4**

`'(* (+ 1'`
`'      (- 23)'`
`'      (* 4 5.6))'`
`'   10)'`
	`Pair('*', Pair(Pair('+', ...)))`
	printed as
	`(* (+ 1 (- 23) (* 4 5.6)) 10)`
		4

Lines forming a Scheme expression	**A number or a Pair with an operator as its first element**	**A number**

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.

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

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, ')', ')'
```

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, ')', ')'
```

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

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

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

Syntactic Analysis

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

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

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

Base case: symbols and numbers

Recursive call: scheme_read sub-expressions and combine them

Syntactic Analysis

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

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

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

Base case: symbols and numbers

Recursive call: scheme_read sub-expressions and combine them

Syntactic Analysis

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

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

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

Base case: symbols and numbers

Recursive call: scheme_read sub-expressions and combine them

Syntactic Analysis

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

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

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

Base case: symbols and numbers

Recursive call: scheme_read sub-expressions and combine them

Syntactic Analysis

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

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

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

Base case: symbols and numbers

Recursive call: scheme_read sub-expressions and combine them

Demo (http://inst.eecs.berkeley.edu/~cs61a/fa12/projects/scalc/scheme_reader.py.html)

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 are evaluated directly.

Evaluation

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

- Primitive expressions 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 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 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 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 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

Demo

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 == '+':
        return ...
```

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 == '+':
        return ...
```

Dispatch on operator name

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 == '+':
        return ...
    if operator == '-':
        ...
    ...
```

Dispatch on operator name

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 == '+':
        return ...
    if operator == '-':
        ...
    ...
```

Dispatch on operator name

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.

Demo

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 == '-':
```

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 == '-':
        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 == '-':
        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 == '-':
        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 == '-':
        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 == '-':
        if len(args) == 0:
            raise TypeError(operator + ' requires at least 1 argument')
        ...
    ...
    if operator == '/':
```

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 == '-':
        if len(args) == 0:
            raise TypeError(operator + ' requires at least 1 argument')
        ...
    ...
    if operator == '/':
        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 == '-':
        if len(args) == 0:
            raise TypeError(operator + ' requires at least 1 argument')
        ...
    ...
    if operator == '/':
        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 == '-':
        if len(args) == 0:
            raise TypeError(operator + ' requires at least 1 argument')
        ...
    ...
    if operator == '/':
        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.

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.

Demo

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