Announcements

- Hog revisions due tonight
- HW10 due Wednesday
- Last chance to fill out survey on Piazza
  - We need to schedule alternate final exam times for those who have a conflict, so if you do, let us know on the survey when you are available
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.
Computers have software written in many different languages.

Machine languages: statements can be interpreted by hardware

• All data are represented as a sequence 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 a sequence 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 a sequence 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 a sequence of bits
• All statements are primitive instructions

High-level languages: hide concerns about those details
• Primitive data types beyond just bits
Computers have software written in many different languages

Machine languages: statements can be interpreted by hardware
• All data are represented as a sequence of bits
• All statements are primitive instructions

High-level languages: hide concerns about those details
• Primitive data types beyond just bits
• Statements/expressions, data can be non-primitive (e.g. calls)
Computers have software written in many different languages

Machine languages: statements can be interpreted by hardware

- All data are represented as a sequence of bits
- All statements are primitive instructions

High-level languages: hide concerns about those details

- Primitive data types beyond just bits
- Statements/expressions, data 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 a sequence of bits
- All statements are primitive instructions

High-level languages: hide concerns about those details
- Primitive data types beyond just bits
- Statements/expressions, data 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 a sequence of bits
• All statements are primitive instructions

High-level languages: hide concerns about those details
• Primitive data types beyond just bits
• Statements/expressions, data 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
Computers have software written in many different languages

Machine languages: statements can be interpreted by hardware
- All data are represented as a sequence of bits
- All statements are primitive instructions

High-level languages: hide concerns about those details
- Primitive data types beyond just bits
- Statements/expressions, data 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
Computers have software written in many different languages

Machine languages: statements can be interpreted by hardware
• All data are represented as a sequence of bits
• All statements are primitive instructions

High-level languages: hide concerns about those details
• Primitive data types beyond just bits
• Statements/expressions, data 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: 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: 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: 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: 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: 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

\[
\begin{align*}
\text{> } & ( + \ (*) \ 3 \ 5 \ (- \ 10 \ 6) ) \\
\text{19} \\
\text{> } & ( + \ (*) \ 3 \\
& \quad ( + \ (*) \ 2 \ 4 \\
& \quad \quad ( + \ 3 \ 5)) \\
& \quad ( + \ (- \ 10 \ 7) \\
& \quad \quad 6)) \\
\text{57}
\end{align*}
\]
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)
Reading Scheme Lists
Reading Scheme Lists

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

(\langle element_0 \rangle \langle element_1 \rangle \ldots \langle element_n \rangle)
Reading Scheme Lists

A Scheme list is written as elements in parentheses:

\((\langle element_0 \rangle \ \langle element_1 \rangle \ \ldots \ \langle element_n \rangle)\)

A recursive Scheme list
A Scheme list is written as elements in parentheses:

\[(\text{element}_0, \text{element}_1, \ldots, \text{element}_n)\]
A Scheme list is written as elements in parentheses:

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

A recursive Scheme list
A Scheme list is written as elements in parentheses:

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

Each \text{element} can be a combination or primitive.
A Scheme list is written as elements in parentheses:

\[
(\langle\text{element}_0\rangle \ \langle\text{element}_1\rangle \ ... \ \langle\text{element}_n\rangle)
\]

A recursive Scheme list

Each \langle\text{element}\rangle can be a combination or primitive

\[
(+ \ (* \ 3 \ (+ \ (* \ 2 \ 4) \ (+ \ 3 \ 5))) \ (+ \ (- \ 10 \ 7) \ 6))
\]
A Scheme list is written as elements in parentheses:

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

Each \textit{<element>} can be a combination or primitive

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

The task of \textit{parsing} a language involves coercing a string representation of an expression to the expression itself.
Reading Scheme Lists

A Scheme list is written as elements in parentheses:

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

Each \texttt{<element>} can be a combination or primitive

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

The task of \textit{parsing} a language involves coercing a string representation of an expression to the expression itself

parsers must validate that expressions are well-formed
A Scheme list is written as elements in parentheses:

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

A recursive Scheme list

Each \textit{<element>} can be a combination or primitive

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

The task of \textit{parsing} a language involves coercing a string representation of an expression to the expression itself

Parsers must validate that expressions are well-formed

(http://inst.eecs.berkeley.edu/~cs61a/sp13/projects/scalc/scheme_reader.py.html)
Parsing
A parser takes a sequence of lines and returns an expression.
A parser takes a sequence of lines and returns an expression.
Parsing

A parser takes a sequence of lines and returns an expression.
A parser takes a sequence of lines and returns an expression.
A parser takes a sequence of lines and returns an expression.
A parser takes a sequence of lines and returns an expression.

\[
\begin{align*}
  & (+ 1) \\
  & (- 23) \\
  & (* 4 5.6))
\end{align*}
\]
A parser takes a sequence of lines and returns an expression.

\[((+ 1')( - 23')( (*) 4 5.6))\]
A parser takes a sequence of lines and returns an expression.

Parser steps:
- **Lexical analysis**
- **Syntactic analysis**

Example lines:

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

Example tokens:

```
(', '+', 1
', '('-', 23)
', '*', '4', '5.6'))
```
A parser takes a sequence of lines and returns an expression.

Lexical analysis

Syntactic analysis
A parser takes a sequence of lines and returns an expression.
A parser takes a sequence of lines and returns an expression.

Lexical analysis: 

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

Syntactic analysis:

'(' +', 1
'(', '-', 23, ')'

expression
A parser takes a sequence of lines and returns an expression.
A parser takes a sequence of lines and returns an expression.

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

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

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

Lines expression: 
- Parsing 
- '+' 
- '1'
- '(- 23)' 
- '(* 4 5.6)'

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

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

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

Cal
A parser takes a sequence of lines and returns an expression.

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

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

- Iterative process
A parser takes a sequence of lines and returns an expression.

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

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

- **Iterative process**
- **Checks for malformed tokens**
A parser takes a sequence of lines and returns an expression.

- Iterative process
- Checks for malformed tokens
- Determines types of tokens
A parser takes a sequence of lines and returns an expression.

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

A parser takes a sequence of lines and returns an expression.

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

- **Syntactic analysis**

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

( ', '+', 1
  ( ', '-' 23, ')'  
  ( ', '* 4, 5.6, ')' ), ')' 
```
A parser takes a sequence of lines and returns an expression.

- Iterative process
- Checks for malformed tokens
- Determines types of tokens
- Processes one line at a time
A parser takes a sequence of lines and returns an expression.

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

Parsing

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

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

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

printed as

```
(+ 1 (- 23) (* 4 5.6))
```
A parser takes a sequence of lines and returns an expression.

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

Pair("+", Pair(1, ...))

print as
(+ 1 (- 23) (* 4 5.6))
A parser takes a sequence of lines and returns an expression.

- **Lexical analysis**
  - Parses the input into tokens
  - Converts the input into a stream of tokens

- **Syntactic analysis**
  - Interprets the tokens to form a syntactically correct expression
  - Uses a parser tree to represent the structure of the expression

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

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

- **Tree-recursive process**
  - Balances parentheses
A parser takes a sequence of lines and returns an expression.

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

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

Expressions:
- Parsing `( + 1 ' ( - 23 ) ' ( * 4 5.6 ) ' )` is printed as `Pair( '+', Pair(1, ...) )`
- `( + 1 ( - 23 ) ( * 4 5.6 ) )`
A parser takes a sequence of lines and returns an expression.

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

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

Example:

- Original input: \((+ 1' (- 23)' (* 4 5.6))\)
- Lexical analysis: \('(+, '+', 1, '-', 23, '(:'. *, '*, 4, 5.6, ')', ')')\)
- Syntactic analysis: Pair('+', Pair(1, ...))
- Printed as: \((+ 1 (- 23) (* 4 5.6))\)
Syntactic Analysis
Syntactic Analysis

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

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

Each call to \texttt{scheme\_read} consumes the input tokens for exactly one expression.

\begin{verbatim}
'( '+', 1, '(' '-', 23, ')' , '(' '*', 4, 5.6, ')' , ')
\end{verbatim}

\textbf{Base case}: symbols and numbers

\textbf{Recursive call}: \texttt{scheme\_read} sub-expressions and combine them as pairs
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 as pairs
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 as pairs

([Link to scheme_reader.py](http://inst.eecs.berkeley.edu/~cs61a/sp13/projects/scalc/scheme_reader.py.html))
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*
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

'(+ 2 2)'
Expression Trees

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

Parser | expression | Evaluator

'( + 2 2)'
A basic interpreter has two parts: a parser and an *evaluator*

\[
\begin{align*}
\text{Expression Trees} \\
\text{Parser} & \quad \text{Evaluator} \\
\text{Pair}(+), \ \text{Pair}(2, \ \text{Pair}(2, \ \text{nil}))
\end{align*}
\]
A basic interpreter has two parts: a parser and an *evaluator*.

\[
\text{'(+ 2 2)'} \quad \text{Pair('+', Pair(2, Pair(2, nil))}
\]
Expression Trees

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

'(+ 2 2)' → \text{Pair}(+', \text{Pair}(2, \text{Pair}(2, \text{nil}))') → 4
Expression Trees

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

`'( + 2 2)'`

Parser: `Pair('+', Pair(2, Pair(2, nil)))`
Value: 4

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

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

- **Parser**: Parses the expression. For example, `'+22'` becomes `Pair('+', Pair(2, Pair(2, nil)))`
  - `4`

- **Evaluator**: Evaluates the expression. For example, `'(1+(-23))*(45.6)10'`
  - `-20`
  - `printed as (* (+1(-23)(* 45.6))10)`
Expression Trees

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

(lines) '(+ 2 2)' [Parser] Pair('+', Pair(2, Pair(2, nil))) [Evaluator] value 4

Parser

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

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

printed as

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

value 4
Expression Trees

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

Parser

Evaluator

Lines forming a Scheme expression

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

'(* (+ 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.

Parser

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

Evaluator

- Lines
- Expression
- Value

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

- '(* (+ 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.

Parser:

- '(+ 2 2)'
  - Pair('+', Pair(2, Pair(2, nil)))
  - Value: 4

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

Evaluator:

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

- Value:
  - A number

Diagram:

- Lines
- Parser
- Expression
- Evaluator
- Value

Diagram shows the process of parsing and evaluating Scheme expressions.
Expression Trees

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

Parser

Evaluator

A number or a Pair with an operator as its first element

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

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

Examples:

1. `'( + 2 2)'`
   - `Pair('+', Pair(2, Pair(2, nil)))`
   - 4

2. `'(* (+ 1'
   
   '   ( - 23)'
   
   '   (* 4 5.6))'
   
   '   10))'
   - `Pair('*', Pair(Pair('+', ...)))
     - `printed as`
     - `(* (+ 1 (- 23) (* 4 5.6)) 10)`
   - 4

These are the lines that form a Scheme expression.
Evaluation

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

```python
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

```python
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

```python
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

```python
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

```python
def calc_apply(operator, args):
    """Apply the named operator to a list of args."""
    if operator == '+':
        return ...
    if operator == '-':
        ...
    ...
```

(https://inst.eecs.berkeley.edu/~cs61a/sp13/projects/scalc/scalc.py.html)
Raising Application Errors
Raising Application Errors

The – and / operators have restrictions on argument number
Raising Application Errors

The – and / operators have restrictions on argument number

Raising exceptions in *apply* can identify such issues
Raising Application Errors

The – and / operators have restrictions on argument number

Raising exceptions in *apply* can identify such issues

```python
def calc_apply(operator, args):
```
Raising Application Errors

The – and / operators have restrictions on argument number

Raising exceptions in *apply* can identify such issues

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

The – and / operators have restrictions on argument number

Raising exceptions in *apply* can identify such issues

```python
def calc_apply(operator, args):
    """Apply the named operator to a list of args."""
    if operator == '-':
```
Raising Application Errors

The – and / operators have restrictions on argument number

Raising exceptions in apply can identify such issues

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

The – and / operators have restrictions on argument number.

Raising exceptions in apply can identify such issues:

```python
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 – and / operators have restrictions on argument number.

Raising exceptions in \textit{apply} can identify such issues.

\begin{verbatim}

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')

    ...
\end{verbatim}
Raising Application Errors

The – and / operators have restrictions on argument number.

Raising exceptions in apply can identify such issues.

```python
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 – and / operators have restrictions on argument number

Raising exceptions in apply can identify such issues

```python
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 – and / operators have restrictions on argument number.

Raising exceptions in `apply` can identify such issues.

```python
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 – and / 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')
    ...

Read-Eval-Print Loop
The user interface to many programming languages is an interactive loop, which
The user interface to many programming languages is an interactive loop, which

- Reads an expression from the user,
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,
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,
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

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

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

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