Reading Scheme Lists

A scheme list is written as elements in parentheses:

```
(element_0 element_1 ... element_n)
```

Each element can be a combination or primitive

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

The task of parsing a language involves coercing a string representation of an expression to the expression itself.

[Demo](http://composingprograms.com/examples/scalc/scheme_reader.py.html)

**Parsing**

A Parser takes text and returns an expression

<table>
<thead>
<tr>
<th>Text</th>
<th>Lexical Analysis</th>
<th>Syntactic Analysis</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>'(' ' ' ')'</td>
<td>'(' ' ' ')'</td>
<td>'(' ' ' ')'</td>
<td>'(' ' ' ')'</td>
</tr>
<tr>
<td>' (+ 1 '   ' (- 23)'   ' (* 4 5.6))'</td>
<td>S</td>
<td>S</td>
<td>S</td>
</tr>
</tbody>
</table>

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

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

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

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

**Calculator Syntax**

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

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

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

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

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

**Calculator Semantics**

The value of a calculator expression is defined recursively.

**Primitive:** A number evaluates to itself.

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

+ Left of the arguments
+ Product of the arguments
- If one argument, negate it. If more than one, subtract the rest from the first.
/ If one argument, invert it. If more than one, divide the rest from the first.

Expressions are represented as Scheme lists: Source code is data.
The Eval Function

The eval function computes the value of an expression, which is always a number. It is a generic function that dispatches on the type of the expression (primitive or call).

### Implementation

```python
def calc_eval(exp):
    if type(exp) in (int, float):
        return exp
    elif isinstance(exp, Pair):
        arguments = exp.second.map(calc_eval)
        return calc_apply(exp.first, arguments)
    else:
        raise TypeError
```

### Language Semantics

<table>
<thead>
<tr>
<th>Evaluation Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A number evaluated</td>
<td>A number evaluated to itself.</td>
</tr>
<tr>
<td>A call expression</td>
<td>A call expression evaluated to its argument value combined by an operator.</td>
</tr>
</tbody>
</table>

### Applying Built-in Operators

The apply function applies some operation to a (Scheme) list of argument values:

### Implementation

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

### Language Semantics

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Sum of the arguments.</td>
</tr>
<tr>
<td>-</td>
<td>Difference of the arguments.</td>
</tr>
<tr>
<td>*</td>
<td>Product of the arguments.</td>
</tr>
<tr>
<td>/</td>
<td>Quotient of the arguments.</td>
</tr>
</tbody>
</table>

### Interactive Interpreters

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

1. Print a prompt
2. Read user input from the user
3. Parse the text input into an expression
4. Evaluate the expression
5. If any errors occur, report those errors, otherwise
6. Print the value of the expression and repeat.

### Raising Exceptions

Exceptions are raised within lexical analysis, syntactic analysis, eval, and apply:

#### Example exceptions

- **Lexical analysis**: The token 2.3.4 raises ValueError("invalid numeral")
- **Syntactic analysis**: An extra ) raises SyntaxError("unexpected token")
- **Eval**: An empty combination raises TypeError("() is not a number or call expression")
- **Apply**: No arguments to - raises TypeError("- requires at least 1 argument")

### Handling Exceptions

An interactive interpreter prints information about each error so that the user has an opportunity to try again in the current environment.

The Structure of an Interpreter

- **Base cases**
  - Primitive values (numbers)
  - Lookup values bound to symbols
- **Recursive calls**
  - Eval(operator, operands) of call expressions
  - Apply(procedure, arguments)
  - Eval(body) of special forms
- **Special cases**
  - Built-in procedures
  - Execution of user-defined procedures

The Structure of an Interpreter

- **Base cases**
  - Primitive values (numbers)
  - Look-up values bound to symbols
- **Recursive calls**
  - Eval(operator, operands) of call expressions
  - Apply(procedure, arguments)
  - Eval(body) of special forms
- **Special cases**
  - Built-in procedures
  - Execution of user-defined procedures
Scheme Evaluation

The `scheme_eval` function chooses behavior based on expression form:
- Symbols are looked up in the current environment
- Self-evaluating expressions are returned as values
- All other legal expressions are represented as Scheme lists, called combinations

\[
\text{if} \quad \text{<predicate>} \quad \text{<consequent>} \quad \text{<alternative>}
\]

\[
\text{define} \quad \text{<name>} \quad \text{<expression>}
\]

\[
\text{lambda} \quad \text{<formal-parameters>} \quad \text{<body>}
\]

\[
\text{<operator>} \quad \text{<operand 0>} \ldots \quad \text{<operand k>}
\]

Logical Forms

Logical Special Forms

Logical forms may only evaluate some sub-expressions:
- If expression: \( (\text{if} \quad \text{<predicate>} \quad \text{<consequent>} \quad \text{<alternative>}) \)
- And \( \text{and} \quad \text{<e1>} \ldots \quad \text{<en>} \)
- Or \( \text{or} \quad \text{<e1>} \ldots \quad \text{<en>} \)
- Cond expression: \( (\text{cond} \quad \text{<p1>} \quad \text{<e1>}) \ldots \quad (\text{<pn>} \quad \text{<en>}) \quad \text{(else} \quad \text{<e>}) \)

The value of an if expression is the value of a sub-expression:
- Evaluate the predicate
- Choose a sub-expression: <consequent> or <alternative>
- Evaluate that sub-expression to get the value of the whole expression

Quotation

The quote special form evaluates to the quoted expression, which is not evaluated:

\[
\text{quote} \quad \text{<expression>}
\]

The `<expression>` itself is the value of the whole quote expression

- `<expression>` is shorthand for `quote <expression>`
- `(quote (+ 1 2))` evaluates to the three-element Scheme list`'(1 2)`

Lambda Expressions

Lambda expressions evaluate to user-defined procedures:

\[
\text{lambda} \quad \text{<formal-parameters>} \quad \text{<body>}
\]

\[
\text{lambda} \quad \text{(x)} \quad (*) \quad x \quad x
\]

Frames and Environments

A frame represents an environment by having a parent frame:

Frames are Python instances with methods `lookup` and `define`

In Project 4, Frames do not hold return values

Define Expressions

A scheme list of symbols

A scheme list of expressions

A Frame instance
Define Expressions

Define binds a symbol to a value in the first frame of the current environment.

(define <name> <expression>)

1. Evaluate the <expression>
2. Bind <name> to its value in the current frame

Procedure definition is shorthand of define with a lambda expression

(define <name> (lambda (<formal parameters>) <body>))

Applying User-Defined Procedures

To apply a user-defined procedure, create a new frame in which formal parameters are bound to argument values, whose parent is the env attribute of the procedure

Evaluate the body of the procedure in the environment that starts with this new frame

(define demo (if (null? s) '(3) (cons (car s) (demo (cdr s)))))

.demo (list 1 2)