**Reading Scheme Lists**

A Scheme list is written as elements in parentheses:

```
(scheme_list_element_1 scheme_list_element_2 ... scheme_list_element_n)
```

Each `scheme_list_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:

```
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>Tokens</th>
<th>Syntactic Analysis</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>(1 2 3)</code></td>
<td>Pair <code>1</code>, <code>2</code>, <code>3</code></td>
<td><code>(1 (2 3))</code></td>
<td><code>Pair(1, Pair(2, Pair(3, Nil)))</code></td>
<td><code>1 2 3</code></td>
</tr>
<tr>
<td><code>(1 2 . 3)</code></td>
<td><code>1</code> . <code>2</code></td>
<td><code>1 2</code></td>
<td><code>Pair(1, Pair(2, Pair(3, Nil)))</code></td>
<td><code>Pair(1, 2)</code></td>
</tr>
</tbody>
</table>

### Syntactic Analysis

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

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

**Base case:** symbols and numbers

**Recursive call:** `scheme_read` sub-expressions and combine them.

```
(len (Pair (1 (2 3))))
```

**Traceback:**

```
Traceback (most recent call last):
  ... TypeError: length attempted on improper list
```

### The Pair Class

The Pair class represents Scheme pairs and lists. A list is a pair whose second element is either a list or nil.

```python
class Pair:
    """A Pair has two instance attributes:
    first and second.
    For a Pair to be a well-formed list,
    second is either a well-formed list or nil.
    Some methods only apply to well-formed lists.
    """
    def __init__(self, first, second):
        self.first = first
        self.second = second
```

```
>>> = Pair(1, Pair(2, Pair(3, nil)))
>>> print(s)
(1 2 3)
>>> len(s)
3
>>> print(Pair(1, 2))
(1 . 2)
>>> print(Pair(1, Pair(2, 3)))
(1 2 . 3)
>>> len(Pair(1, Pair(2, 3)))
Traceback (most recent call last):
  ... TypeError: length attempted on improper list
```

Scheme expressions are represented as Scheme lists! Source code in data
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.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Expression Tree</th>
<th>Representation as Pairs</th>
</tr>
</thead>
<tbody>
<tr>
<td>(+ 3 4 5)</td>
<td>[+ 3 4 5]</td>
<td>([+ 3 4 5])</td>
</tr>
<tr>
<td>(3 + 5)</td>
<td>[3 + 5]</td>
<td>([3 + 5])</td>
</tr>
</tbody>
</table>

Calculator Semantics

The value of a calculator expression is defined recursively.

Primitive: A number evaluates to itself.

Call: A call expression evaluates to its argument values combined by an operator.
- Sum of the arguments
- Product of the arguments
- If one argument, negate it. If more than one, subtract the rest from the first.
- If one argument, invert it. If more than one, divide the rest from the first.

Expression Tree

<table>
<thead>
<tr>
<th>Expression</th>
<th>Expression Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>(+ 5 (* 2 3) (* 2 5 5))</td>
<td>(+ 5 (* 2 3) (* 2 5 5))</td>
</tr>
</tbody>
</table>

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).

Evaluation

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

Applying Built-in Operators

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

In calculator, all operations are named by built-in operators: +, -, *, /

<table>
<thead>
<tr>
<th>Implementation</th>
<th></th>
<th>Language Semantics</th>
</tr>
</thead>
</table>
| def calc_apply(operator, args):
  if operator == '+':
    return reduce(add, args, 0)
  elif operator == '-':
    ...  
  elif operator == '*':
    ...  
  elif operator == '/':
    ...  
  else:
    raise TypeError |

Interactive Interpreters

<table>
<thead>
<tr>
<th>Implementation</th>
<th></th>
<th>Language Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>calc_apply('+', [1, 2, 3])</td>
<td>6</td>
<td>Sum of the arguments</td>
</tr>
<tr>
<td>calc_apply('-', [10, 20, 30])</td>
<td>50</td>
<td>Subtract the rest from the first</td>
</tr>
<tr>
<td>calc_apply('*', [10, 20, 30])</td>
<td>600</td>
<td>Product of the arguments</td>
</tr>
<tr>
<td>calc_apply('/', [10, 20, 30])</td>
<td>0.5</td>
<td>Divide the rest from the first</td>
</tr>
</tbody>
</table>

(Demo)
Read-Eval-Print Loop

The user interface for many programming languages is an interactive interpreter
1. Print a prompt
2. Read text 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("expected 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

A well-designed interactive interpreter should not halt completely on an error, so that the user has an opportunity to try again in the current environment

Interpreting Scheme

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(sub-expressions) of special forms

Requires an environment for symbol lookup
Creates a new environment when applying procedure is applied

Special Forms
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

\[
\text{if} \langle \text{predicate} \rangle \langle \text{consequent} \rangle \langle \text{alternative} \rangle
\]

\[
\text{define} \langle \text{name} \rangle \langle \text{expression} \rangle
\]

\[
\text{lambda} \langle \text{formal-parameters} \rangle \langle \text{body} \rangle
\]

\[
\langle \text{operator} \rangle \langle \text{operand 0} \rangle \ldots \langle \text{operand k} \rangle
\]

Special forms are identified by the first list element. Any combination that is not a known special form is a call.

\[
\text{define} \; \text{(demo s)} \; \left( \text{if} \left( \text{null?} \; s \right) \; '3 \; \left( \text{cons} \; \text{(car s)} \; \left( \text{demo} \; \text{(cdr s)} \right)\right) \right)
\]

Logical Forms

Logical Special Forms

Logical forms may only evaluate some sub-expressions:

- IF expression: \( (\text{if} \langle \text{predicate} \rangle \langle \text{consequent} \rangle \langle \text{alternative} \rangle) \)
- AND and OR:
  \( (\text{and} \langle e1 \rangle \ldots \langle en \rangle) \),
  \( (\text{or} \langle e1 \rangle \ldots \langle en \rangle) \)
- Cond expression:
  \( (\text{cond} \langle p1 \rangle \langle e1 \rangle \ldots \langle pn \rangle \langle en \rangle \langle \text{else} \rangle \langle e \rangle) \)

The value of an IF expression is the value of a sub-expression:

- Evaluate the predicate
- Choose a sub-expression: \( \langle \text{consequent} \rangle \) or \( \langle \text{alternative} \rangle \)
- 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:

\[
\text{quote} \langle \text{expression} \rangle \quad \text{quote} \left( + 1 2 \right)
\]

\[
\langle \text{expression} \rangle \text{ itself is the value of the whole quote expression}
\]

\[
\langle \text{expression} \rangle \text{ is shorthand for } (\text{quote} \langle \text{expression} \rangle)
\]

\[
(\text{quote} \left( + 1 2 \right)) \text{ is equivalent to } '\left( + 1 2 \right)
\]

Lambda Expressions

The scheme_read parser converts shorthand ` to a combination that starts with quote:

\[
(\text{quote})
\]
Lambda Expressions

Lambda expressions evaluate to user-defined procedures:

\[
\lambda(formal-parameters) \rightarrow \text{body}
\]

\[
\lambda(x) \rightarrow (x \cdot x)
\]

```python
class LambdaProcedure:
    def __init__(self, formals, body, env):
        self.formals = formals
        self.body = body
        self.env = env
```

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

Define binds a symbol to a value in the current environment.

\[
\text{define} \quad \text{name} \rightarrow \text{expression}
\]

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

#### Procedure Definition

Procedure definition is shorthand of `define` with a lambda expression.

\[
\text{define} \quad \text{name} \rightarrow \lambda(formal \text{ parameters}) \rightarrow \text{body}
\]

Define is an abbreviation for `define` with a lambda expression.

#### Eval/Apply in Lisp 1.5

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 return attribute of the procedure.

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