Parsing
Reading Scheme Lists

A Scheme list is written as elements in parentheses:

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

A Scheme list

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
A Parser takes text and returns an expression.

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

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

**Syntactic analysis**
- Pair('+', Pair(1, ...))

**Expression**
- Pair('+', Pair(1, ...))
- printed as
- (+ 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

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

(Demo)
Scheme-Syntax Calculator

(Demo)
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

>>> s = 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 is data

(Demo)
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.

Expression | Expression Tree | Representation as Pairs
---|---|---
(* 3 (+ 4 5) (* 6 7 8)) | ![Expression Tree](image) | ![Representation as Pairs](image)
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.

```
13 (+ 5 (* 2 3) (* 2 5 5))
```
Evaluation
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**

- **A number evaluates**...
  - to itself
- **A call expression evaluates**...
  - to its argument values combined by an operator
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>Language Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>def calc_apply(operator, args):</td>
<td></td>
</tr>
<tr>
<td>if operator == '+':</td>
<td></td>
</tr>
<tr>
<td>return reduce(add, args, 0)</td>
<td>*:</td>
</tr>
<tr>
<td>elif operator == '-':</td>
<td>*:</td>
</tr>
<tr>
<td>...</td>
<td>Sum of the arguments</td>
</tr>
<tr>
<td>elif operator == '*':</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>elif operator == '/':</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>else:</td>
<td></td>
</tr>
<tr>
<td>raise TypeError</td>
<td></td>
</tr>
<tr>
<td>(Demo)</td>
<td></td>
</tr>
</tbody>
</table>
Interactive Interpreters
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

(Demo)
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")

(Demo)
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.

(Demo)
Interpreting Scheme
The Structure of an Interpreter

**Eval**
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

**Apply**
Base cases:
- Built-in primitive procedures

Recursive calls:
- Eval(body) of user-defined procedures

Requires an environment for symbol lookup
Creates a new environment each time a user-defined 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

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

(demo (list 1 2))
```
Logical Forms
Logical Special Forms

Logical forms may only evaluate some sub-expressions

- **If** expression: \((\text{if} \ <\text{predicate}> \ <\text{consequent}> \ <\text{alternative}>)\)
- **And** and **or**: \((\text{and} \ <\text{e1}> ... <\text{en}>)\), \((\text{or} \ <\text{e1}> ... <\text{en}>)\)
- **Cond** expression: \((\text{cond} \ (<p1> <\text{e1}>) ... (<pn> <\text{en}>) \ (\text{else} <\text{e}>)\))

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

- Evaluate the predicate
- Choose a sub-expression: \(<\text{consequent}>\) or \(<\text{alternative}>\)
- Evaluate that sub-expression to get the value of the whole expression

(Demo)
Quotation
Quotation

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

\[
(\text{quote } \langle \text{expression} \rangle) \quad \text{(quote } (+ 1 2)) \quad \text{evaluates to the three-element Scheme list} \\
(\text{quote } \langle \text{expression} \rangle) \quad \text{evaluates to} \\
(+ 1 2)
\]

The \langle expression \rangle itself is the value of the whole quote expression

'\langle expression \rangle is shorthand for (quote \langle expression \rangle)

\[
(\text{quote } (1 2)) \quad \text{is equivalent to} \quad '(1 2)
\]

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

(Demo)
Lambda Expressions
Lambda Expressions

Lambda expressions evaluate to user-defined procedures

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

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

class LambdaProcedure:

def __init__(self, formals, body, env):
    self.formals = formals  # A scheme list of symbols
    self.body = body        # A scheme list of expressions
    self.env = env          # A Frame instance
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.

```
g: Global frame
   y: 3
   z: 5

f1: [parent=g]
   x: 2
   z: 4
```

(Demo)
Define Expressions
Define Expressions

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

\[
\text{(define } \text{name} \text{ expression})
\]

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

\[
\text{(define } x \text{ (+ 1 2))}
\]

Procedure definition is shorthand of define with a lambda expression

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

\[
\text{(define name (lambda (formal parameters) body))}
\]
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 s) (if (null? s) '(3) (cons (car s) (demo (cdr s)))))
```

```
(demo (list 1 2))
```
Eval/Apply in Lisp 1.5

\[
\text{apply\hspace{1mm}[\text{fn};x;a]} = \\
\text{[atom\hspace{1mm}[\text{fn}]} \rightarrow \text{[eq\hspace{1mm}[\text{fn};\text{CAR}] \rightarrow \text{caar\hspace{1mm}[x]}]; \\
\text{eq\hspace{1mm}[\text{fn};\text{CDR}] \rightarrow \text{cdar\hspace{1mm}[x]}]; \\
\text{eq\hspace{1mm}[\text{fn};\text{CONS}] \rightarrow \text{cons\hspace{1mm}[\text{car\hspace{1mm}[x];cadr\hspace{1mm}[x]}]; \\
\text{eq\hspace{1mm}[\text{fn};\text{ATOM}] \rightarrow \text{atom\hspace{1mm}[\text{car\hspace{1mm}[x]}]; \\
\text{eq\hspace{1mm}[\text{fn};\text{EQ}] \rightarrow \text{eq\hspace{1mm}[\text{car\hspace{1mm}[x];cadr\hspace{1mm}[x]}]; \\
\text{T} \rightarrow \text{apply\hspace{1mm}[\text{eval\hspace{1mm}[\text{fn};a];x;a]}]; \\
\text{eq\hspace{1mm}[\text{car\hspace{1mm}[\text{fn}];LAMBDA}] \rightarrow \text{eval\hspace{1mm}[\text{caddr\hspace{1mm}[\text{fn}];pair\hspace{1mm}\text{lis\hspace{1mm}[\text{cadr\hspace{1mm}[\text{fn}];x;a]}]; \\
\text{eq\hspace{1mm}[\text{car\hspace{1mm}[\text{fn}];LABEL}] \rightarrow \text{apply\hspace{1mm}[\text{caddr\hspace{1mm}[\text{fn}];x;cons\hspace{1mm}[\text{cons\hspace{1mm}[\text{cadr\hspace{1mm}[\text{fn}]; \\
\text{caddr\hspace{1mm}[\text{fn}]}}];a]}]; \\
\text{eval\hspace{1mm}[\text{e};a]} = \text{[atom\hspace{1mm}[\text{e}] \rightarrow \text{cdr\hspace{1mm}[\text{assoc\hspace{1mm}[\text{e};a]}]; \\
\text{atom\hspace{1mm}[\text{car\hspace{1mm}[\text{e}]}]} \rightarrow \\
\text{[eq\hspace{1mm}[\text{car\hspace{1mm}[\text{e}];QUOTE}] \rightarrow \text{cadr\hspace{1mm}[\text{e}]]; \\
\text{eq\hspace{1mm}[\text{car\hspace{1mm}[\text{e}];COND}] \rightarrow \text{evcon\hspace{1mm}[\text{cdr\hspace{1mm}[\text{e}];a}]; \\
\text{T} \rightarrow \text{apply\hspace{1mm}[\text{car\hspace{1mm}[\text{e}];evlis\hspace{1mm}[\text{cdr\hspace{1mm}[\text{e}];a};a]}]; \\
\text{T} \rightarrow \text{apply\hspace{1mm}[\text{car\hspace{1mm}[\text{e}];evlis\hspace{1mm}[\text{cdr\hspace{1mm}[\text{e}];a};a]}]]}
\]