Scheme: variant of LISP

LISP: John McCarthy (1958)
Scheme: Steele and Sussman (1973)

Those who do not learn from history are doomed to repeat it. - George Santayana

Scheme (now called Racket)

- Scheme = LISP + ALGOL
  - symbolic list manipulation
  - block structure; static scoping
- Symbolic Computation
  - Translators
    - Parsers, Compilers, Interpreters.
  - Reasoners
    - Natural language understanding systems
    - Database querying
  - Data / Text Processors
    - emacs

“Striking” Features

- Simple and uniform syntax
- Support for convenient list processing
- (Automatic) Garbage Collection
- Environment for Rapid Prototyping
- Intrinsically generic functions
- Dynamic typing (flexible but inefficient)
- Compositionality through extensive use of lists. (minimizing impedance mismatch)

Expressions

- Literals
- Variables
- Procedure calls

- Literals
  - numerals(2), strings("abc"), boolean(#t), etc.
- Variables
  - Identifier represents a variable. Variable reference denotes the value of its binding.
Scheme Identifiers

- E.g., `y`, `x`, `+`, `two+two`, `zero?`, `list->string`, etc
- (Illegal) `5x`, `y2`, `ab c`, etc

- Identifiers
  - reserved keywords
  - Not case sensitive
  - variables
    - pre-defined functions/variables
    - ordinary
  - `functions = procedures`

Procedure Call (application)

- `(operator-expr operand-expr ...)
  - prefix expression (proc/op arg1 arg2 arg3 ...)
  - `(+ x (p 2 3))`
  - `Function value`
  - `( (f 2 3) 5 6)`

- `f is Higher-order function`

Simple Scheme Expressions

- `( + 12 13 ) = 25`
- `( / 12 14 ) = 0.8571428571428571`
- `( + 2.2+1.1i 2.2+1.1i ) = 4.4+2.2i`
- `( * 2.2+1.1i 0+1i ) = (-1.1+2.2i)`
- `( < 2.2 3 4.4 5 ) = #t`
- `( + 25 ) = #t`
- `( negative? (- 1 0) ) = #f`
- `( expt 2 10 ) = 1024`
- `( sqrt 144 ) = 12`
- `( string->number "12" 8 ) = 10`
- `( string->number "AB" 16 ) = 171`

Scheme Evaluation Rule (REPL)

- Expression
  - blank separated, parentheses delimited, nested list structure
- Evaluation
  - Evaluate each element of the outerlist recursively.
  - Apply the result of the operator expression (of function type) to the results of zero or more operand expressions.
Simple Example

\[(+ (\times 2 \ 3) (- 4 (/ 6 3)) )\]

\[(+ (\times 2 \ 3) (- 4 (/ 6 3)) )\]

\[(+ (\times 2 \ 3) (- 4 \ 2) )\]

\[ (+ 6 \ 2) \]

\[ 8 \]

Lists

- Ordered sequence of elements of arbitrary type (Heterogeneous)
  - Empty List: \[
  ()
  \]
  - 3-element list: \[
  (a \ b \ 3)
  \]
  - Nested list: \[
  (1 \ (2.3 \ x) \ 4)
  \]

Sets vs lists

- Duplication matters: \[(a) \neq (a \ a)\]
- Order matters: \[(a \ b) \neq (b \ a)\]
- Nesting-level matters: \[(a) \neq ((a))\]

Key Point

- Syntax of Scheme programs has been deliberately designed to match syntax of lists -- its most prominent data structure
- Important consequence:
  Supports meta-programming
  - Enables program manipulating programs

Special Forms

- Definition
  \[ (\text{define} \ <\text{var}> \ <\text{expr}>) \]
  > \[
  (\text{define} \ \text{false} \ \#f)
  \]

- Conditional
  \[ (\text{if} \ <\text{test}> \ <\text{then}> \ <\text{else}>) \]
  > \[
  (\text{if} \ (\text{symbol?} \ 'a) \ (\text{zero?} \ 5) \ (/ \ 10 \ 0))
  \]
Symbols

- Identifiers treated as *primitive values*.
  - Distinct from identifiers that name variables in program text.
  - Distinct from strings (sequence of characters).

- Some meta-programming primitives
  ```scheme```
  ```quote
  symbol?
  ```

Symbolic Data: **quote** function

- "say *your name* aloud"
- "say ‘*your name*’ aloud"

variable vs symbolic data (to be taken literally)
- `(define x 2)`
- `- x` = 2
- `(quote x)` = x
- `'x` = x
- `(+ 3 6)` = 9
- `'(+ 3 6)` -> list value

Pairs

```scheme```
```(cons 'a 'b)
```
```Printed as: (a . b)
```

```scheme```
```(cons 'a (cons 'b '()))
```
```Printed as: (a b)
```

(cont’d)

```scheme```
```(cons 'a (cons 'a (cons 'b '())))
```
```Printed as: (a a b)
```
List Functions

- Operations
  - car, cdr, cons, null?, ...
  - list, append, ...
  - cadr, caddr, caaar, ...

- Expressions
  - `(length (quote (quote a)))` = 2
  - `(length 'quote)`
  - `(cadar X) = (car (cdr (car X)))`

(define xl '(a b c))
{ allocate storage for the list and initialize xl }

- car, first : list -> element
  - `(car xl)` = a

- cdr : list -> list
  - `(cdr xl)` = (b c)
  { non-destructive }

- cons : element x list -> list
  - `(cons 'a '(b c))` = (a b c)

• For all non-empty lists xl:
  `(cons (car xl) (cdr xl)) = xl`

• For all x and lists xl:
  `(car (cons x xl)) = x`
  `(cdr (cons x xl)) = xl`

  - car : first element of the outermost list.
  - cdr : list that remains after removing car.
    `(cons '() '(0))` = ??
    `(cons '() '(0))` = (0)

  - null? : list -> boolean
    `(null? '())` = #t
    `(null? '(a))` = #f
    `(null? 25)` = #f

  - list : elements x ... -> list
    `(list 'a '(a) 'ab)` = (a (a) ab)

  - append : list x ... -> list
    `(append '() (list 'a) '(0))` = (a 0)
    { variable arity functions }
### Role of parentheses

<table>
<thead>
<tr>
<th>Program</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extra call to interpreter</td>
<td>Extra level of nesting</td>
</tr>
</tbody>
</table>

Program:
- `(list 'append)`
  - `(append)`
- `(list (append))` = `(()

Data:
- `(car '(a))` = `(a)`
- `(car '((a)))` = `(a)`

### Equivalence Test

```
(eq? (cons 3 '()) (cons 3 '())) = #f
(define a (cons 3 '()))
(define b (cons 3 '()))
(eq? a b) = #f
(define c a)
(eq? a c) = #t
```

### Equivalent Scheme Expressions

```
(car (list append list))
   (cons 'a ')' '(1 2 3) )
= ( append ''(a) '(1 2 3) )
= '(a 1 2 3)

(cdr (cons car cdr))
   (cons 'car 'cdr) )
= ( cdr '(car . cdr) )
= 'cdr
```