Scheme : *variant of LISP*

LISP : John McCarthy

Scheme : Steele and Sussman

Scheme

- 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

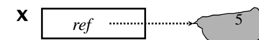
- Simple and uniform syntax
- Support for convenient list processing
- (Automatic) Garbage Collection
- Environment for Rapid Prototyping

- 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,  x5,  +,  two+two,`
  `zero?,  list->string,` etc
- (*Illegal*) `5x, y)2, ab c,` etc
- Identifiers
  - reserved keywords
  - variables
    - pre-defined functions/constants
    - ordinary
- *functions = procedures*

Not case sensitive

Procedure Call (application)

- `(operator-expr  operand-expr ...)`
  - **prefix** expression (proc/op arg1 arg2 arg3 ...)
- Order of evaluation of the sub-expressions is "explicitly" left *unspecified* by Scheme.
  - cf. C is silent about it.
  - cf. Java specifies a *left to right* processing.

```
(+   x  (p   2   3))
((f  2  3)   5   6)
```


Simple Scheme Expressions

- `(+ 12 13)`
  `= 25`
- `(+ 12 13)`
  `= 1.0`
- `(+ 2.2+1.1i 2.2+1.1i )`
  `= 4.4+2.2i`
- `(* 2.2+1.1i 0+i )`
  `= (-1.1+2.2i)`
- `(< 2.2 3 4.4 5)`
  `= #t`

- `(positive? 25)`
  `= #t`
- `(negative? (- 1 0))`
  `= #f`
- `(expt 2 10)`
  `= 1024`
- `(sqrt 144)`
  `= 12`
- `(string->number "12" 8)`
  `= 10`
- `(string->number "ABC")`
  `= #f`

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.

Lists

- Ordered sequence of elements of arbitrary type (*Heterogeneous*)
  - *Empty List* : `()`
  - *3-element list :* `(a b 3)`
  - *Nested list* : `(1 (2.3 x) 4)`
  - `(a) =/= (a a)`
  - `(a b) =/= (b a)`
  - `( a ) =/= ( ( a ) )`
- Supports meta-programming
  - program manipulating programs

Special Forms

- *Definition*

```
(define    <var>    <expr>)
```

- *Conditional*

```
(if  <test>  <then>  <else>)
    > (define false  #f)
    > (if (symbol? 'a)
              (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

```
quote
symbol?
```


Symbolic Data : `quote` function

"say *your name* aloud"

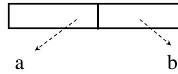
"say **'your name'** aloud"

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

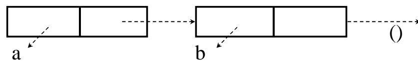
Pairs

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



(a . (b . ()))

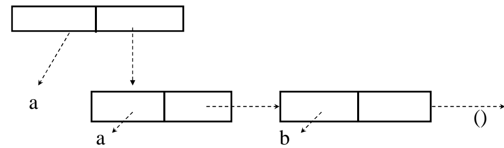
`(cons 'a (cons 'b '()) )` *Printed as:* (a b)



(cont'd)

`(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)))
  - (length '''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  '()  '())          = ??
(cons  '() '())           = (())
```


- `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

- **Program**
  - Extra call to interpreter

```
(list 'append)
     = (append)

(list (append))
     = (())
```

- **Data**
  - Extra level of nesting

```
(car '(a))
     = a

(car '((a)))
     = (a)
```

```
(list (append)
      (append))  = ?
   = (()())
```

```
(list append
      append)  = ?
   = (@fn @fn)
```


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
```