Lists

In Lisp and Scheme lists are a special, widely-used form of S-Expressions.

() represents the empty or null list

(A) represents the list containing A. By definition, (A) \equiv (A . ())

(A B) represents the list containing A and B. By definition,

(A B) \equiv (A . (B . ()))

In general, (A B C ... Z) \equiv

(A . (B . (C . ... (Z . ()) ...)))

(A B C) \equiv

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>()</td>
</tr>
</tbody>
</table>

Function Calls

In List and Scheme, function calls are represented as lists.

(A B C) means:
Evaluate A (to a function)
Evaluate B and C (as parameters)
Call A with B and C as its parameters
Then use the value returned by the call as the “meaning” of (A B C).

cons, car and cdr are predefined symbols bound to built-in functions
that build and access lists and S-Expressions.

Literals (of type integer, real, rational, complex, string, character and
boolean) evaluate to themselves.

For example (⇒ means “evaluates to”)

\(\text{cons 1 2) } \Rightarrow (1 . 2)\)

\(\text{cons 1 () } \Rightarrow (1)\)

\(\text{car (cons 1 2)) } \Rightarrow 1\)

\(\text{cdr (cons 1 ())) } \Rightarrow ()\)

But,

\(\text{car (1 2)) } \text{ fails during execution!}\)

Why?
The expression (1 2) looks like a call, but 1 isn't a function! We need some way to “quote” symbols and lists we don’t want evaluated.

(quote arg)
is a special function that returns its argument unevaluated.

Thus (quote (1 2)) doesn’t try to evaluate the list (1 2); it just returns it.

Since quotation is so often used, it may be abbreviated using a single quote. That is

\(\text{quote arg) } \equiv '\text{arg}\)

Thus

\(\text{car ' (a b c)) } \Rightarrow a\)

\(\text{cdr ' ( (A) (B) (C)) } \Rightarrow\)

\(\text{( (B) (C) )}\)

\(\text{cons 'a '1) } \Rightarrow (a . 1)\)

But,

\(\text{'cdr ' (A B)) } \text{ fails!}\)

Why?
**User-defined Functions**

The list

\( \text{(lambda (args) (body))} \)

evaluates to a function with \( \text{(args)} \) as its argument list and \( \text{(body)} \) as the function body.

No quotes are needed for \( \text{(args)} \) or \( \text{(body)} \).

Thus

\( \text{(lambda (x) (+ x 1))} \) evaluates to the increment function.

Similarly,

\( \text{((lambda (x) (+ x 1)) 10)} \Rightarrow 11 \)

Since functions are frequently defined, we may abbreviate

\( \text{(define id (lambda (args) (body)))} \)

as

\( \text{(define (id args) (body))} \)

Thus

\( \text{(define (plus1 x) (+ x 1))} \)

We can bind values and functions to global symbols using the **define** function.

The general form is

\( \text{(define id object)} \)

\( \text{id} \) is not evaluated but \( \text{object} \) is. \( \text{id} \) is bound to the value \( \text{object} \) evaluates to.

For example,

\( \text{(define pi 3.1415926535)} \)

\( \text{(define plus1 (lambda (x) (+ x 1)))} \)

\( \text{(define pi*2 (* pi 2))} \)

Once a symbol is defined, it evaluates to the value it is bound to:

\( \text{(plus1 12)} \Rightarrow 13 \)

**Conditional Expressions in Scheme**

A **predicate** is a function that returns a boolean value. By convention, in Scheme, predicate names end with “?”

For example,

\( \text{number? symbol? equal? null? list?} \)

In conditionals, \#\( \text{f} \) is false, and everything else, including \#t, is true.

The **if** expression is

\( \text{(if pred E1 E2)} \)

First \( \text{pred} \) is evaluated. Depending on its value (\#\( \text{f} \) or not), either \( \text{E1} \) or \( \text{E2} \) is evaluated (but not both) and returned as the value of the **if** expression.
For example,

```lisp
(if (= 1 (+ 0 1))
  'Yes
  'No)
```

```lisp
(define
  (fact n)
  (if (= n 0)
      1
      (* n (fact (- n 1))))
)
```

---

**Generalized Conditional**

This is similar to a switch or case:

```lisp
(cond
  (p1  e1)
  (p2  e2)
  ...
  (else  en)
)
```

Each of the predicates (p1, p2, ...) is evaluated until one is true (≠ #f). Then the corresponding expression (e1, e2, ...) is evaluated and returned as the value of the cond. else acts like a predicate that is always true.

Example:

```lisp
(cond
  ((= a 1) 2)
  ((= a 2) 3)
  (else 4)
)
```