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, \((A B C \ldots Z) \equiv (A . (B . (C . \ldots (Z . (()) \ldots )))\)

\((A B C ) \equiv \)

```
   +---+
   |   |
   +---+---+
     |   |
   +---+---+
        |   |
   +---+---+---+
             |   |
   +---+---+---+---+
                  |   |
   +---+---+---+---+---+
                      |   |
   +---+---+---+---+---+---+
                          |   |
   +---+---+---+---+---+---+---+
```

A  B  C  ()
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 \((A \ B \ C)\).

\(\text{cons, car and cdr}\) 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”)

(cons 1 2) ⇒ (1 . 2)

(cons 1 () ) ⇒ (1)

(car (cons 1 2)) ⇒ 1

(cdr (cons 1 ())) ⇒ ()

But,

(car (1 2)) 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

\[
(quote \ arg) \equiv \ 'arg
\]

Thus

\[
(car \ '\ (a \ b \ c)) \Rightarrow a
\]

\[
(cdr \ '\ (\ (A) \ (B) \ (C))) \Rightarrow \ (\ (B) \ (C) )
\]

\[
(cons \ 'a \ '1) \Rightarrow (a . \ 1)
\]

But,

\[(\ 'cdr \ '\ (A \ B))\] fails!

Why?
User-defined Functions

The list

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

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,

\(((\text{lambda} \ (x) \ (+ \ x \ 1)) \ 10) \Rightarrow 11\)
We can bind values and functions to global symbols using the `define` function.

The general form is

```
(define id object)
```

`id` is not evaluated but `object` is. `id` is bound to the value `object` evaluates to.

For example,

```
(define pi 3.1415926535)
(define plus1 (lambda (x) (+ x 1)))
(define pi*2 (* pi 2))
```

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

```
(plus1 12) ⇒ 13
```
Since functions are frequently defined, we may abbreviate

\[
\text{(define id}
  \quad \text{(lambda (args) (body)) )}
\]

as

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

Thus

\[
\text{(define (plus1 x) (+ x 1))}
\]
Conditional Expressions in Scheme

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

For example,

number? symbol? equal? null? list?

In conditionals, #f is false, and everything else, including #t, is true. The if expression is

(if pred E1 E2)

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

\[
\text{(if} \ (= \ 1 \ (+ \ 0 \ 1)) \\
\text{ 'Yes} \\
\text{ 'No} \\
\text{)}
\]

\[
\text{(define} \\
\text{ (fact n)} \\
\text{ (if} \ (= \ n \ 0) \\
\text{ 1} \\
\text{ (*} \ n \ (\text{fact} \ (- \ n \ 1))) \\
\text{)}
\]

)
Generalized Conditional

This is similar to a switch or case:

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

Each of the predicates \((p_1, p_2, \ldots)\) is evaluated until one is true \((\neq \#f)\). Then the corresponding expression \((e_1, e_2, \ldots)\) is evaluated and returned as the value of the \texttt{cond}. \texttt{else} acts like a predicate that is always true.

Example:

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