

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

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 (\Rightarrow means “evaluates to”)

`(cons 1 2) \Rightarrow (1 . 2)`

`(cons 1 ()) \Rightarrow (1)`

`(car (cons 1 2)) \Rightarrow 1`

`(cdr (cons 1 ())) \Rightarrow ()`

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 often needed, 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

`(lambda (args) (body))`

evaluates to a function with `(args)` as its argument list and `(body)` as the function body.

No quotes are needed for `(args)` or `(body)`.

Thus

`(lambda (x) (+ x 1))` evaluates to the increment function.

Similarly,

`((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

```
(define id  
  (lambda (args) (body)))
```

as

```
(define (id args) (body))
```

Thus

```
(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,

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

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

Generalized Conditional

This is similar to a switch or case.

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

Each of the predicates (**p1**, **p2**, ...) is evaluated until one is true ($\neq \#f$). 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:

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