LISP - SET

Common Lisp does not provide a set data type. However, it provides number of functions that allows set operations to be performed on a list.

You can add, remove, and search for items in a list, based on various criteria. You can also perform various set operations like: union, intersection, and set difference.

Implementing Sets in LISP

Sets, like lists are generally implemented in terms of cons cells. However, for this very reason, the set operations get less and less efficient the bigger the sets get.

The **adjoin** function allows you to build up a set. It takes an item and a list representing a set and returns a list representing the set containing the item and all the items in the original set.

The **adjoin** function first looks for the item in the given list, if it is found, then it returns the original list; otherwise it creates a new cons cell with its **car** as the item and **cdr** pointing to the original list and returns this new list.

The **adjoin** function also takes **:key** and **:test** keyword arguments. These arguments are used for checking whether the item is present in the original list.

Since, the adjoin function does not modify the original list, to make a change in the list itself, you must either assign the value returned by adjoin to the original list or, you may use the macro **pushnew** to add an item to the set.

Example

Create a new source code file named main.lisp and type the following code in it.

```
; creating myset as an empty list
(defparameter *myset* ())
(adjoin 1 *myset*)
(adjoin 2 *myset*)

; adjoin did not change the original set
;so it remains same
(write *myset*)
(terpri)
(setf *myset* (adjoin 1 *myset*))
(setf *myset* (adjoin 2 *myset*))

;now the original set is changed
(write *myset*)
(terpri)

;adding an existing value
(pushnew 2 *myset*)

;no duplicate allowed
(write *myset*)
(terpri)

;pushing a new value
(pushnew 3 *myset*)
(write *myset*)
(terpri)
```

When you execute the code, it returns the following result:

```
NIL
(2 1)
```

```
(2 1)
(3 2 1)
```

Checking Membership

The member group of functions allows you to check whether an element is member of a set or not.

The following are the syntaxes of these functions:

```
member item list &key :test :test-not :key
member-if predicate list &key :key
member-if-not predicate list &key :key
```

These functions search the given list for a given item that satisfies the test. If no such item is found, then the functions returns **nil.** Otherwise, the tail of the list with the element as the first element is returned.

The search is conducted at the top level only.

These functions could be used as predicates.

Example

Create a new source code file named main.lisp and type the following code in it.

```
(write (member 'zara '(ayan abdul zara riyan nuha)))
(terpri)
(write (member-if #'evenp '(3 7 2 5/3 'a)))
(terpri)
(write (member-if-not #'numberp '(3 7 2 5/3 'a 'b 'c)))
```

When you execute the code, it returns the following result:

```
(ZARA RIYAN NUHA)
(2 5/3 'A)
('A 'B 'C)
```

Set Union

The union group of functions allows you to perform set union on two lists provided as arguments to these functions on the basis of a test.

The following are the syntaxes of these functions:

```
union list1 list2 &key :test :test-not :key
nunion list1 list2 &key :test :test-not :key
```

The **union** function takes two lists and returns a new list containing all the elements present in either of the lists. If there are duplications, then only one copy of the member is retained in the returned list.

The **nunion** function performs the same operation but may destroy the argument lists.

Example

Create a new source code file named main.lisp and type the following code in it.

```
(setq set1 (union '(a b c) '(c d e)))
(setq set2 (union '(#(a b) #(5 6 7) #(f h))
   '(#(5 6 7) #(a b) #(g h)) :test-not #'mismatch)
)

(setq set3 (union '(#(a b) #(5 6 7) #(f h))
   '(#(5 6 7) #(a b) #(g h)))
```

```
)
(write set1)
(terpri)
(write set2)
(terpri)
(write set3)
```

When you execute the code, it returns the following result:

```
(A B C D E)
(#(F H) #(5 6 7) #(A B) #(G H))
(#(A B) #(5 6 7) #(F H) #(5 6 7) #(A B) #(G H))
```

Please Note

The union function does not work as expected without **:test-not #'mismatch** arguments for a list of three vectors. This is because, the lists are made of cons cells and although the values look same to us apparently, the **cdr** part of cells does not match, so they are not exactly same to LISP interpreter/compiler. This is the reason; implementing big sets are not advised using lists. It works fine for small sets though.

Set Intersection

The intersection group of functions allows you to perform intersection on two lists provided as arguments to these functions on the basis of a test.

The following are the syntaxes of these functions:

```
intersection list1 list2 &key :test :test-not :key
nintersection list1 list2 &key :test :test-not :key
```

These functions take two lists and return a new list containing all the elements present in both argument lists. If either list has duplicate entries, the redundant entries may or may not appear in the result.

Example

Create a new source code file named main.lisp and type the following code in it.

```
(setq set1 (intersection '(a b c) '(c d e)))
(setq set2 (intersection '(#(a b) #(5 6 7) #(f h))
   '(#(5 6 7) #(a b) #(g h)) :test-not #'mismatch)
)

(setq set3 (intersection '(#(a b) #(5 6 7) #(f h))
   '(#(5 6 7) #(a b) #(g h)))
)
(write set1)
(terpri)
(write set2)
(terpri)
(write set3)
```

When you execute the code, it returns the following result:

```
(C)
(#(A B) #(5 6 7))
NIL
```

The nintersection function is the destructive version of intersection, i.e., it may destroy the original lists.

Set Difference

The set-difference group of functions allows you to perform set difference on two lists provided as arguments to these functions on the basis of a test.

The following are the syntaxes of these functions:

```
set-difference list1 list2 &key :test :test-not :key
nset-difference list1 list2 &key :test :test-not :key
```

The set-difference function returns a list of elements of the first list that do not appear in the second list.

Example

Create a new source code file named main.lisp and type the following code in it.

```
(setq set1 (set-difference '(a b c) '(c d e)))
(setq set2 (set-difference '(#(a b) #(5 6 7) #(f h))
   '(#(5 6 7) #(a b) #(g h)) :test-not #'mismatch)
)
(setq set3 (set-difference '(#(a b) #(5 6 7) #(f h))
   '(#(5 6 7) #(a b) #(g h)))
)
(write set1)
(terpri)
(write set2)
(terpri)
(write set3)
```

When you execute the code, it returns the following result:

```
(A B)
(#(F H))
(#(A B) #(5 6 7) #(F H))
```