LISP - SEQUENCES

Sequence is an abstract data type in LISP. Vectors and lists are the two concrete subtypes of this data type. All the functionalities defined on sequence data type are actually applied on all vectors and list types.

In this section, we will discuss most commonly used functions on sequences.

Before starting on various ways of manipulating sequences *i. e. , vectorsandlists*, let us have a look at the list of all available functions.

Creating a Sequence

The function make-sequence allows you to create a sequence of any type. The syntax for this function is:

```
make-sequence sqtype sqsize &key :initial-element
```

It creates a sequence of type *sqtype* and of length *sqsize.*

You may optionally specify some value using the *:initial-element* argument, then each of the elements will be initialized to this value.

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

```
(write (make-sequence '(vector float)
   10
   :initial-element 1.0))
```

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

```
#(1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0)
```

Generic Functions on Sequences

Function	Description
elt	It allows access to individual elements through an integer index.
length	It returns the length of a sequence.
subseq	It returns a sub-sequence by extracting the subsequence starting at a particular index and continuing to a particular ending index or the end of the sequence.
copy-seq	It returns a sequence that contains the same elements as its argument.
fill	It is used to set multiple elements of a sequence to a single value.
replace	It takes two sequences and the first argument sequence is destructively modified by copying successive elements into it from the second argument sequence.
count	It takes an item and a sequence and returns the number of times the item appears in the sequence.
reverse	It returns a sequence contains the same elements of the argument but in reverse order.
nreverse	It returns the same sequence containing the same elements as sequence but in reverse order.
concatenate	It creates a new sequence containing the concatenation of any number of

	sequences.
position	It takes an item and a sequence and returns the index of the item in the sequence or nil.
find	It takes an item and a sequence. It finds the item in the sequence and returns it, if not found then it returns nil.
sort	It takes a sequence and a two-argument predicate and returns a sorted version of the sequence.
merge	It takes two sequences and a predicate and returns a sequence produced by merging the two sequences, according to the predicate.
map	It takes an n-argument function and n sequences and returns a new sequence containing the result of applying the function to subsequent elements of the sequences.
some	It takes a predicate as an argument and iterates over the argument sequence, and returns the first non-NIL value returned by the predicate or returns false if the predicate is never satisfied.
every	It takes a predicate as an argument and iterate over the argument sequence, it terminates, returning false, as soon as the predicate fails. If the predicate is always satisfied, it returns true.
notany	It takes a predicate as an argument and iterate over the argument sequence, and returns false as soon as the predicate is satisfied or true if it never is.
notevery	It takes a predicate as an argument and iterate over the argument sequence, and returns true as soon as the predicate fails or false if the predicate is always satisfied.
reduce	It maps over a single sequence, applying a two-argument function first to the first two elements of the sequence and then to the value returned by the function and subsequent elements of the sequence.
search	It searches a sequence to locate one or more elements satisfying some test.
remove	It takes an item and a sequence and returns the sequence with instances of item removed.
delete	This also takes an item and a sequence and returns a sequence of the same kind as the argument sequence that has the same elements except the item.
substitute	It takes a new item, an existing item, and a sequence and returns a sequence with instances of the existing item replaced with the new item.
nsubstitute	It takes a new item, an existing item, and a sequence and returns the same sequence with instances of the existing item replaced with the new item.
mismatch	It takes two sequences and returns the index of the first pair of mismatched elements.

Standard Sequence Function Keyword Arguments

Argument	Meaning	Default Value
:test	It is a two-argument function used to compare item $orvalueextractedby: keyfunction$ to element.	EQL
:key	One-argument function to extract key value from actual sequence element. NIL means use element as is.	NIL
:start	Starting index $inclusive$ of subsequence.	0

:end	Ending index *exclusive* of subsequence. NIL indicates end of sequence.	NIL
:from-end	If true, the sequence will be traversed in reverse order, from end to start.	NIL
:count	Number indicating the number of elements to remove or substitute or NIL to indicate all *REMOVEandSUBSTITUTEonly*.	NIL

We have just discussed various functions and keywords that are used as arguments in these functions working on sequences. In the next sections, we will see how to use these functions using examples.

Finding Length and Element

The **length** function returns the length of a sequence, and the **elt** function allows you to access individual elements using an integer index.

Example

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

```
(setq x (vector 'a 'b 'c 'd 'e))
(write (length x))
(terpri)
(write (elt x 3))
```

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

```
5
D
```

Modifying Sequences

Some sequence functions allows iterating through the sequence and perform some operations like, searching, removing, counting or filtering specific elements without writing explicit loops.

The following example demonstrates this:

Example 1

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

```
(write (count 7 '(1 5 6 7 8 9 2 7 3 4 5)))
(terpri)
(write (remove 5 '(1 5 6 7 8 9 2 7 3 4 5)))
(terpri)
(write (delete 5 '(1 5 6 7 8 9 2 7 3 4 5)))
(terpri)
(write (substitute 10 7 '(1 5 6 7 8 9 2 7 3 4 5)))
(terpri)
(write (find 7 '(1 5 6 7 8 9 2 7 3 4 5)))
(terpri)
(write (position 5 '(1 5 6 7 8 9 2 7 3 4 5)))
```

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

```
2
(1 6 7 8 9 2 7 3 4)
(1 6 7 8 9 2 7 3 4)
(1 5 6 10 8 9 2 10 3 4 5)
7
1
```

Example 2

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

```
(write (delete-if #'oddp '(1 5 6 7 8 9 2 7 3 4 5)))
(terpri)
(write (delete-if #'evenp '(1 5 6 7 8 9 2 7 3 4 5)))
(terpri)
(write (remove-if #'evenp '(1 5 6 7 8 9 2 7 3 4 5) :count 1 :from-end t))
(terpri)
(setq x (vector 'a 'b 'c 'd 'e 'f 'g))
(fill x 'p :start 1 :end 4)
(write x)
```

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

```
(6 8 2 4)
(1 5 7 9 7 3 5)
(1 5 6 7 8 9 2 7 3 5)
#(A P P P E F G)
```

Sorting and Merging Sequences

The sorting functions take a sequence and a two-argument predicate and return a sorted version of the sequence.

Example 1

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

```
(write (sort '(2 4 7 3 9 1 5 4 6 3 8) #'<))
(terpri)
(write (sort '(2 4 7 3 9 1 5 4 6 3 8) #'>))
(terpri)
```

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

```
(1 2 3 3 4 4 5 6 7 8 9)
(9 8 7 6 5 4 4 3 3 2 1)
```

Example 2

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

```
(write (merge 'vector #(1 3 5) #(2 4 6) #'<))
(terpri)
(write (merge 'list #(1 3 5) #(2 4 6) #'<))
(terpri)
```

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

```
#(1 2 3 4 5 6)
(1 2 3 4 5 6)
```

Sequence Predicates

The functions every, some, notany, and notevery are called the sequence predicates.

These functions iterate over sequences and test the Boolean predicate.

All these functions takes a predicate as the first argument and the remaining arguments are sequences.

Example

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

```
(write (every #'evenp #(2 4 6 8 10)))
(terpri)
(write (some #'evenp #(2 4 6 8 10 13 14)))
(terpri)
(write (every #'evenp #(2 4 6 8 10 13 14)))
(terpri)
(write (notany #'evenp #(2 4 6 8 10)))
(terpri)
(write (notevery #'evenp #(2 4 6 8 10 13 14)))
(terpri)
```

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

```
T
T
NIL
NIL
T
```

Mapping Sequences

We have already discussed the mapping functions. Similarly the **map** function allows you to apply a function on to subsequent elements of one or more sequences.

The **map** function takes a n-argument function and n sequences and returns a new sequence after applying the function to subsequent elements of the sequences.

Example

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

```
(write (map 'vector #'* #(2 3 4 5) #(3 5 4 8)))
```

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

```
#(6 15 16 40)
```