Sequences, lists, vectors, strings

- The name lisp comes from list processing, representation of information using lists is key aspect of the language
- Sequence is the most general list-like type
- Lists are a subtype of sequences
- Vectors are a subtype of arrays
- Strings are a subtype of vectors
- Functions designed for sequences work on all four
- Functions designed for arrays also work on vectors, strings
- Functions designed for vectors also work on strings
- Efficiency best if you use the most specific for your type

Basic syntax

- Basic syntax looks like a function, e.g. (x 10 “foo” 3.5), will need to come up with a way to distinguish them
- It’s ok to have empty lists, ()
- Nil and empty list are treated as the same thing (so empty list also acts as false for boolean expressions)
- It’s ok to have lists inside lists, e.g. (1 2 (x y z)) and can be as deeply nested as desired, e.g. (“foo” (1.1 (((5 6) 4)(3 1))))
- Can use listp, stringp, vectorp to check type

Quotes and sequences/lists/etc

- Need a way to tell the difference between a list and a function call, e.g. (x y z) looks like either
- Assumed to be a function call unless told otherwise, e.g. (1 2 3) is assumed to be calling function 1 on parameters 2 and 3
- The list function returns a list, e.g. (list 1 2 3)
- The quote function tells lisp to treat as data, not a function call, e.g. (quote (1 2 3)) does not treat (1 2 3) as a function call
- Putting a single quote in front acts same as quote function, e.g. ‘(1 2 3) means treat it as data, not a function call

Common lisp programming errors

- Putting the function name in front of the bracket (e.g. f(x) instead of (f x)
- missing/extra/misplaced brackets (use a good editor)
- forgetting to type-check parameters or user input
- putting a quote in front of something you intended to be a function call
- forgetting a quote in front of something you meant to be a list of data

Quotes in other contexts

- The single quote, or the quote function, are often used to tell lisp not to evaluate something, e.g.
- ‘x means the symbol x, don’t evaluate as the variable x or function name x

Sequence functions

- (length S) returns length of sequence S
- (elt S i) returns ith element of S
- (setf (elt S i) x) sets ith element to x
- (count e S) counts how often e appears in S
- (remove e S) remove all e’s from S
- (copy-seq S) returns a duplicate of S
- (sort S OP) sorts s using OP for comparison, e.g. (sort S ‘<)
- (concatenate TYPE S1 S2) returns concatenation of sequences S1 and S2, type can be ‘list, ‘string, ‘vector

Some list functions

- (cons e L) returns list that has e as first element, followed by contents of L
- (car L) returns front element of L (crashes if L empty)
- (cdr L) returns list of all of L except front element
- (nth N L) returns Nth element of L
- (last L) returns last element of L
- (member e L) true iff e is in L
- (null L) true iff L is empty list
- (append L1 L2) returns list with L1 elements then L2 elements

Combinations of car and cdr

- (cadr L) like (car (cdr L))
- (cdar L) like (cdr (car L))
- (cddr L) like (cdr (cdr L))
- Etc, can use any sequence of up to 3 a’s and d’s
- Trivia: car came from content-address-register and cdr came from content-decrement-register, from original implementations way back in the 60’s

Some vector functions

- (vector x y z) builds/returns vector with elements x y z
- (svref V i) returns ith element of vector V
- (setf (svref V i) x) sets ith element to x
- And, of course, any of the array and sequence functions are usable on vectors

Programming with lists

- Lots of recursion, using car/cdr to decompose lists into head/tail, or cons to construct one element at a time
- Example: count the elements in a list

```
(defun countElements (L)
   (cond
      ((not (listp L)) nil)
      ((null L) 0)
      (t (+ 1 (countElements (cdr L))))))
```

Example: append

- Want (append L1 L2) to return a list with all the elements of L1, followed by all the elements of L2
- First element of L1 will be front, the rest can be build by appending the rest of L1 with L2

```
(defun append (L1 L2)
    (cond
        ((or (not (listp L1)) (not (listp L2))) nil)
        ((null L1) L2)
        ((null L2) L1)
        (t (cons (car L1) (append (cdr L1) L2)))))
```

Example: reverse a list

- Will use a helper function and build up a list of the elements we’ve reversed so far

```
(defun reverse (L)
   (if (not (listp L)) nil (revHelp L ‘())))

(defun revHelp (LR sofar)
   (cond
      ((null LR) sofar)
      (t (revHelp (cdr LR) (cons (car LR)
sofar)))))
```

Arrays

- Can create multidimensional arrays by specifying the dimensions as a list

```
(make-array ‘(3 4)) ; returns 3 x 4 array
```

- Can initialize elements when we create it, e.g.

```
(make-array ‘(3 4) :initial-element 27)  ; inits to 27
```

- Can lookup dimension list using array-dimensions

```
(array-dimensions myArray)
```

- Can look up the size of just the i’th dimenion

```
(array-dimension myArray i)
```

Accessing array elements

- To look up an element, use aref with indices, e.g.
- (aref myArray i j) ; returns myArray[i][j]
- (setf (aref myArray i j) v) ; like myArray[i][j] = v