LISP BASICS

References:

- Wilensky, chapters 1-4, 6

Jumping right in

A Lisp function:

```
(defun increment (n)
   (+ n 1)
)
```

A call to the function:

```
(increment 21)
```

Pure Functional Programming

- A program consists of function definitions and calls. There is no other structure.
- No loops. Just recursion.
- No side effects — a function may not change its parameters, or do input or output.
- No variables. Just parameters.
- Implicit memory mgmt: no new or free.
- One data structure: the list. Can representing anything, including code.
- Functions can be:
  - passed as an argument;
  - returned from a function;
  - represented by a data structure;
  - and such a data structure can then be evaluated.

More Interesting Examples

Write functions to:

1. Return the sum of the first n integers.
2. Increment every integer in a list.
3. Reverse a list.

Basic Lisp

Pre-defined Functions

- `(+` *–numbers–*`)`: Returns the sum of the numbers.
- Other arithmetic operators: `-`, `*`, `/`
- `(equal` *a, b*`)`: Returns `t` iff *a* and *b* are the same number or character, or are the same Lisp object.
- `(null` *object*`)`: Returns `t` iff *object* is `nil`.
- `(car` *list*`)`: Returns the first element of *list*. Pre: *list* must be a list.
- `(cdr` *list*`)`: Returns *list* with the first element removed. Pre: *list* must be a list.
- `(caddar` *list*`)`: Returns the `car` of the `cdr` of the `cdr` ...

`(cons` *a b*`)`: Returns a list whose `car` is *a* and whose `cdr` is *b*. Pre: *b* must be a list.

`(append` *a b*`)`: Returns the list formed by joining the elements of *a* and *b* together.
Pre: *a* and *b* must be lists.

`(list` *–objects–*`)`: Returns a list of the *objects*.

`(quote` it object`)`: Returns *object* without evaluating it. Same as ‘*object*.

Other things

- `t`: Used for “true” by predicates.
- `nil`: Means “the empty list”. Same as ‘`()`. Used for “false” by predicates.
-
```
(cond  (exp11  exp12  exp13 ...)
       (exp21  exp22  exp23 ...)
                 ...

       (expn1  expn2  expn3 ...)
)
```

The CommonLisp 4.3 Interface

```
dvp.cs 1% cl
Allegro CL 4.3 ...

USER(1): (+ 8 3 5 16 9)
41

USER(2): (defun increment (n)
   (+ n 1)
)
INCREMENT

USER(3): (increment 21)
22

USER(4): :ld egs.cl
; Loading ./egs.cl

USER(5):  (increment_list (1 32 7))
Error: Funcall of 1 which is a non-function.
  [condition type: SIMPLE-ERROR]

[1] USER(6): :reset
```



```
USER(7):  (increment_list ‘(1 32 7))
(2 33 8)

USER(8): (trace increment_list)
(INCREMENT_LIST)

USER(9): (increment_list ‘(1 32 7))
 0: (INCREMENT_LIST (1 32 7))
   1: (INCREMENT_LIST (32 7))
     2: (INCREMENT_LIST (7))
       3: (INCREMENT_LIST NIL)
       3: returned NIL
     2: returned (8)
   1: returned (33 8)
 0: returned (2 33 8)
(2 33 8)

USER(10): (exit)
; Exiting Lisp
dvp.cs %2
```

How Lisp Evaluates

To evaluate an expression, Lisp always does the following:

- Evaluate the operands.
- Apply the operation to the results.

Well, not always. Examples:

These are called **special forms** because they violate the rule.

Why did `(increment 21)` work?