Types, Conditionals, and Loops in Emacs Lisp

What will we cover?

We'll get a sense for the basic data types of the language and how we can use them!

- True and false
- Numbers
- Characters
- Sequences
- Strings
- Lists
- Arrays
- Combining expressions with logic operators
- Conditional logic
- Loops

Follow along with IELM!

You can open up an interactive Emacs Lisp REPL in Emacs by running `M-x ielm`.

I'll be using the following snippet for evaluating code in the REPL:

```
(defun efs/ielm-send-line-or-region ()
  (interactive)
  (unless (use-region-p)
    (forward-line 0)
    (set-mark-command nil)
    (forward-line 1))
  (backward-char 1)
  (let ((text (buffer-substring-no-properties (region-beginning)
                                              (region-end))))
    (with-current-buffer "*ielm*"
      (insert text)
      (ielm-send-input))

    (deactivate-mark)))

(defun efs/show-ielm ()
  (interactive)
  (select-window (split-window-vertically -10))
  (ielm)
  (text-scale-set 1))

(define-key org-mode-map (kbd "C-c C-e") 'efs/ielm-send-line-or-region)
(define-key org-mode-map (kbd "C-c E") 'efs/show-ielm)
```

True and false

Normally a language would have a "boolean" type for expressing "true" and "false".

In Emacs Lisp, we have `t` and `nil` which serve the same purpose.

They are symbols!

```
(type-of t)   ;; symbol
(type-of nil) ;; symbol
```

These symbols are used to provide "yes" and "no" answers for expressions in the language.

There are many "predicates" for the different types which return `t` or `nil`, we will cover them when talking about each type.

Equality

One of the most basic operations you would do on any type is check whether two values are the same!

There are a few ways to do that in Emacs Lisp:

- `eq` - Are the two parameters the same object?
- `eql` - Are the two parameters same object or same number?
- `equal` - Are the two parameters equivalent?
- Type-specific equality predicates

```
(setq test-val '(1 2 3))

(eq 1 1)                  ;; t
(eq 3.1 3.1)              ;; nil
(eq "thing" "thing")      ;; nil
(eq '(1 2 3) '(1 2 3))    ;; nil
(eq test-val test-val)    ;; t

(eql 1 1)                 ;; t
(eql 3.1 3.1)             ;; t
(eql "thing" "thing")     ;; nil
(eql '(1 2 3) '(1 2 3))   ;; nil
(eql test-val test-val)   ;; t

(equal 1 1)               ;; t
(equal 3.1 3.1)           ;; t
(equal "thing" "thing")   ;; t
(equal '(1 2 3) '(1 2 3)) ;; t
(equal test-val test-val) ;; t
```

A general rule is that you should use `equal` for most equality checks or use a type-specific equality check.

Emacs Lisp Manual: Equality Predicates

Numbers

There are two main types of numbers in Emacs Lisp:

- Integers - Whole numbers

- Floating point numbers - Numbers with a decimal

```
1
3.14159

-1
-3.14159

1.
1.0

-0
```

Operations

You can perform mathematical operations on these numbers:

```
(+ 5 5)  ;; 10
(- 5 5)  ;; 0
(* 5 5)  ;; 25
(/ 5 5)  ;; 1

;; Nesting arithmetic!
(* (+ 3 2)
   (- 10 5))  ;; 25

(% 11 5)      ;; 1 - integer remainder
(mod 11.1 5)  ;; 1.099 - float remainder

(1+ 5)   ;; 6
(1- 5)   ;; 4
```

You can also convert between integers and floats:

- `truncate` - Rounds float to integer by moving toward zero
- `round` - Rounds to the nearest integer
- `floor` - Rounds float to integer by subtracting
- `ceiling` - Round up to the next integer

```
(truncate 1.2)   ;; 1
(truncate -1.2)  ;; -1

(floor 1.2)      ;; 1
```

```
(floor -1.2)     ;; -2

(ceiling 1.2)    ;; 2
(ceiling 1.0)    ;; 1

(round 1.5)      ;; 2
(round 1.4)      ;; 1
```

See also:

- Floating point rounding operations
- Bitwise operations
- Standard mathematical functions

Predicates

These predicates will help you identify the number types in code:

```
(integerp 1)     ;; t
(integerp 1.1)   ;; nil
(integerp "one") ;; nil

(floatp 1)       ;; nil
(floatp 1.1)     ;; t
(floatp "one")   ;; nil

(numberp 1)      ;; t
(numberp 1.1)    ;; t
(numberp "one")  ;; nil

(zerop 1)        ;; nil
(zerop 0)        ;; t
(zerop 0.0)      ;; t
```

Comparisons

You can compare two numeric values (even integers against floats):

```
(= 1 1)     ;; t
(= 1 1.0)   ;; t
(= 1 1 1
   1 1 1)   ;; t
```

```
(< 1 2)     ;; t
(> 1 2)     ;; nil
(> 1 1)     ;; nil
(> 1.2 1)   ;; nil

(>= 1 1)     ;; t
(<= -1 -1.0) ;; t

(max 1 5 2 7)  ;; 7
(min -1 3 2 4) ;; -1
```

Characters

Characters are really just integers that are interpreted as characters:

```
?A   ;; 65
?a   ;; 97

?\n  ;; 10
?\t  ;; 9

;; Unicode
?\N{U+E0}    ;; 224
?\u00e0      ;; 224
?\U000000E0  ;; 224
?\N{LATIN SMALL LETTER A WITH GRAVE} ;; 224

;; Control and meta char syntax
?\C-c        ;; 3
(kbd "C-c")  ;; "^C"
?\M-x        ;; 134217848
(kbd "M-x")  ;; [134217848]
```

Emacs Lisp Manual: Character Type

Comparisons

```
(char-equal ?A ?A)
(char-equal ?A 65)
(char-equal ?A ?a)

case-fold-search
```

```
(setq case-fold-search nil)
(setq case-fold-search t)
```

Sequences

In Emacs Lisp, strings, lists, and arrays are all considered sequences

```
(sequencep "Sequence?")     ;; t
(sequencep "")              ;; t

(sequencep '(1 2 3))        ;; t
(sequencep '())             ;; t

(sequencep [1 2 3])         ;; t
(sequencep [])              ;; t

(sequencep 22)              ;; nil
(sequencep ?A)              ;; nil

;; What do you expect?
(sequencep nil)
```

You can get the length of any sequence with `length`:

```
(length "Hello!")    ;; 6
(length '(1 2 3))    ;; 3
(length [5 4 3 2])   ;; 4
(length nil)         ;; 0
```

You can get an element of any sequence at a zero-based index with `elt`:

```
(elt "Hello!" 1)   ;; ?e
(elt "Hello!" -1)  ;; error -out of range

(elt '(3 2 1) 2)  ;; 1
(elt '(3 2 1) 3)  ;; nil - out of range
(elt '(3 2 1) -1)  ;; 3
(elt '(3 2 1) -2)  ;; 3
(elt '(3 2 1) -6)  ;; 3 - seems to always return first element

(elt [1 2 3 4] 2)   ;; 3
```

```
(elt [1 2 3 4] 5)   ;; error - out of range
(elt [1 2 3 4] -1)  ;; error - out of range
```

Strings

Strings are arrays of characters:

```
"Hello!"

"Hello \
 System Crafters!"

"Hello \\ System Crafters!"

(make-string 5 ?!)            ;; !!!!!
(string ?H ?e ?l ?l ?o ?!)    ;; "Hello!"
```

Predicates

```
(stringp "Test!")           ;; t
(stringp 1)                 ;; nil
(stringp nil)               ;; nil

(string-or-null-p "Test")   ;; t
(string-or-null-p nil)      ;; t

(char-or-string-p ?A)       ;; t
(char-or-string-p 65)       ;; t
(char-or-string-p "A")      ;; t

(arrayp "Array?")           ;; t
(sequencep "Sequence?")     ;; t
(listp "List?")             ;; nil
```

Comparisons

You can compare strings for equivalence or for sorting:

- `string=` or `string-equal`
- `string<` or `string-lessp`
- `string>` or `string-greaterp`

```
(string= "Hello" "Hello")    ;; t
(string= "HELLO" "Hello")    ;; nil

(string<  "Hello" "Hello")    ;; nil
(string<  "Mello" "Yello")    ;; t
(string<  "Hell"  "Hello")    ;; t

(string>  "Hello" "Hello")    ;; nil
(string>  "Mello" "Yello")    ;; nil
(string>  "Hell"  "Hello")    ;; nil
```

Emacs Lisp Manual: Text Comparison

Operations

```
(substring "Hello!" 0 4)    ;; Hell
(substring "Hello!" 1)      ;; ello!

(concat "Hello " "System" " " "Crafters" "!")
(concat)

(split-string "Hello System Crafters!")
(split-string "Hello System Crafters!" "s")
(split-string "Hello System Crafters!" "S")

(split-string "Hello System Crafters!" "[ !]")
(split-string "Hello System Crafters!" "[ !]" t)

;; Default splitting pattern is [ \f\t\n\r\v]+

(setq case-fold-search nil)
(setq case-fold-search t)
```

Formatting

You can create a string from existing values using `format`:

```
(format "Hello %d %s!" 100 "System Crafters")
(format "Here's a list: %s" '(1 2 3))
```

There are many more format specifications, mainly for number representations, consult the manual for more info:

Emacs Lisp Manual: Formatting Strings

Writing messages

As you've already seen, you can write messages to the echo area (minibuffer) and `*Messages*` buffer using the `message` function:

```
(message "This is %d" 5)
```

It uses the same formatting specifications as `format!`

Lists

The list is possibly the most useful data type in Emacs Lisp.

Cons Cells

Lists are built out of something called "cons cells". They enable you to chain together list elements using the "cons" container.

You can think of a "cons" like a pair or "tuple" with values that can be accessed with `car` and `cdr`:

- `car` - Get the first value in the cons
- `cdr` - Get the second value in the cons

```
(cons 1 2)  ;; '(1 . 2)
'(1 . 2)    ;; '(1 . 2)

(car '(1 . 2))  ;; 1
(cdr '(1 . 2))  ;; 2

(setq some-cons '(1 . 2))

(setcar some-cons 3)
some-cons             ;; '(3 . 2)

(setcdr some-cons 4)
some-cons             ;; '(3 . 4)
```

Building lists from cons

There are two ways to build a list from cons cells:

```
(cons 1 (cons 2 (cons 3 (cons 4 nil))))
(cons 1 '(2 3 4))

(cons '(1 2 3) '(4 5 6))

(append '(1 2 3) 4)
(append '(1 2 3) '(4))
```

Predicates

```
(listp '(1 2 3))
(listp 1)

(listp nil)       ;; t
(cons 1 nil)
(append '(1) nil)

(listp (cons 1 2))
(listp (cons 1 (cons 2 (cons 3 (cons 4 nil)))))
(consp (cons 1 (cons 2 (cons 3 (cons 4 nil)))))
```

Alists

Association lists (or “alists”) are lists containing cons pairs for the purpose of storing named values:

```
(setq some-alist '((one . 1)
                   (two . 2)
                   (three . 3)))

(alist-get 'one   some-alist)  ;; 1
(alist-get 'two   some-alist)  ;; 2
(alist-get 'three some-alist)  ;; 3
(alist-get 'four  some-alist)  ;; nil

(assq 'one   some-alist)  ;; '(one . 1)
(rassq 1     some-alist)  ;; '(one . 1)
```

```
;; There is no alist-set!
(setf (alist-get 'one some-alist) 5)
(alist-get 'one some-alist)  ;; 5
```

Plists

A property list (or "plist") is another way to do key/value pairs with a flat list:

```
(plist-get '(one 1 two 2) 'one)
(plist-get '(one 1 two 2) 'two)

(plist-put '(one 1 two 2) 'three 3)
```

Arrays

Arrays are sequences of values that are arranged contiguously in memory. They are much faster to access!

The most obvious form of array is a "vector", a list with square brackets. Strings are also arrays!

We know how to access elements in arrays, but you can set them with `aset`:

```
(setq some-array [1 2 3 4])
(aset some-array 1 5)
some-array

(setq some-string "Hello!")
(aset some-string 0 ?M)
some-string
```

We can set all values in an array using `fillarray`

```
(setq some-array [1 2 3])
(fillarray some-array 6)
some-array
```

Logic Expressions

Logic expressions allow you to combine expressions using logical operators (`and` , `or`)

You can think of this as operations on the “truthiness” or “falsiness” of expressions!

What is true?

When evaluating expressions, everything except the value `nil` and the empty list `'()` is considered `t` !

```
(if t        'true 'false)  ;; true
(if 5        'true 'false)  ;; true
(if "Emacs"  'true 'false)  ;; true
(if ""       'true 'false)  ;; true
(if nil      'true 'false)  ;; false
(if '()      'true 'false)  ;; false
```

Logic operators

Emacs provides the following logic operators:

- `not` - Inverts the truth value of the argument
- `and` - Returns the last value if all expressions are truthy
- `or` - Returns the first value that is truthy (short-circuits)
- `xor` - Returns the first value that is truthy (doesn’t short-circuit)

```
(not t)    ;; nil
(not 3)    ;; nil
(not nil)  ;; t

(and t t t t 'foo)   ;; 'foo
(and t t t 'foo t)   ;; 't
(and 1 2 3 4 5)      ;; 5
(and nil 'something) ;; nil

(or  nil 'something) ;; 'something
(or  nil 'something t) ;; 'something
(or (- 3 3) (+ 2 0)) ;; 0
```

Conditional expressions

The `if` expression

As we saw before, the `if` expression evaluates an expression and based on the result, picks one of two "branches" to evaluate next.

The "true" branch is a single expression, the "false" branch can be multiple expressions:

```
(if t 5
  ;; You can add an arbitrary number of forms in the "false" branch
  (message "Doing some extra stuff here")
  (+ 2 2))
```

You can use `progn` to enable multiple expressions in the "true" branch:

```
(if t
  (progn
    (message "Hey, it's true!")
    5)
  ;; You can add an arbitrary number of forms in the "false" branch
  (message "Doing some extra stuff here")
  (+ 2 2))
```

Since this is an expression, it returns the value of the last form evaluated inside of it:

```
(if t 5
  (message "Doing some extra stuff here")
  (+ 2 2))

(if nil 5
  (message "Doing some extra stuff here")
  (+ 2 2))
```

You can use `if` expressions inline when setting variables:

```
(setq tab-width (if (string-equal (format-time-string "%A")
                                  "Monday")
                    3
                    2))
```

The `when` and `unless` expressions

These expressions are useful for evaluating forms when a particular condition is true or false:

- `when` - Evaluate the following forms when the expression evaluates to `t`
- `unless` - Evaluate the following forms when the expression evaluates to `nil`

```
(when (> 2 1) 'foo)    ;; 'foo
(unless (> 2 1) 'foo)  ;; nil

(when (> 1 2) 'foo)    ;; nil
(unless (> 1 2) 'foo)  ;; 'foo
```

Both of these expressions can contain multiple forms and return the result of the last form:

```
(when (> 2 1)
  (message "Hey, it's true!")
  (- 5 2)
  (+ 2 2)) ;; 4

(unless (> 1 2)
  (message "Hey, it's true!")
  (- 5 2)
  (+ 2 2)) ;; 4
```

The `cond` expression

The `cond` expression enables you to concisely list multiple conditions to check with resulting forms to execute:

```
(setq a 1)
(setq a 2)
(setq a -1)

(cond ((eql a 1) "Equal to 1")
      ((> a 1)   "Greater than 1")
      (t         "Something else!"))
```

The `pcase` expression

This one is powerful! We will cover it in a future episode.

Loops

There are 4 ways to loop in Emacs Lisp:

while

Loops until the condition expression returns false:

```
(setq my-loop-counter 0)

(while (< my-loop-counter 5)
  (message "I'm looping! %d" my-loop-counter)
  (setq my-loop-counter (1+ my-loop-counter)))
```

dotimes

```
(dotimes (count 5)
  (message "I'm looping more easily! %d" count))
```

dolist

Loops for each item in a list:

```
(dolist (item '("one" "two" "three" "four" "five"))
  (message "Item %s" item))
```

Recursion

Can be fun and interesting, but not safe for a loop that will have many cycles:

```
(defun efs/recursion-test (counter limit)
  (when (< counter limit)
    (message "I'm looping via recursion! %d" counter)
```

```
      (efs/recursion-test (1+ counter) limit)))

(efs/recursion-test 0 5)
```

What's next?

- Dive into functions!
- Shorter side videos on `pcase`, regular expressions