Integers, floating point numbers, ratios, and characters.

Atoms, as in Prolog or Erlang, not reserved words.

... lambdas.

Sequences are the abstraction above lists, vectors, maps, sets, etc.

They share the methods first, rest, and cons.

The nesting of many ifs is difficult to read.

The special form <code>cond</code> allows as many arguments as needed for if ... then ... elseif ... then ... elseif ...

cond provides :else to catch everything as the final branch.



Instead of returning the atoms false/true, they take advantage of the fact that anything other than false/nil is "true" and return a more meaningful value.

By quoting:

'(a b c)

```
(quote (a b c))
```

Special forms get arguments unevaluated, controlling if/when to evaluate them.

However, special forms are *not* first-class values.

/ which returns a Ratio.

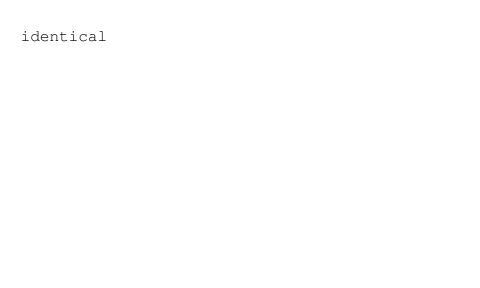
quot performs truncating integer division.

They differ in their handling of signs. If the first and second arguments have different signs, the result of mod will have the same sign as the second argument, while the result of rem will have the same sign as the first argument.

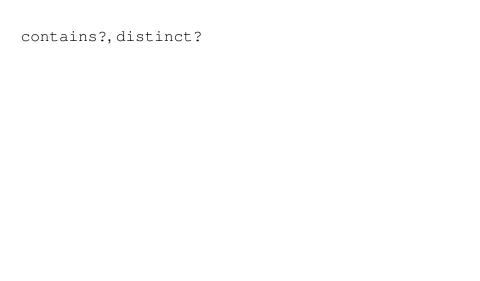
not **and** not= **are provided**.

- == only compares arguments that can be case to java.lang.Number.
- = compares arguments in a type-independent manner. For example, vectors can be equal to lists according to =.

... matters. Forward-references are not allowed.



coll?, seq?, vector?, list?, map?, set?

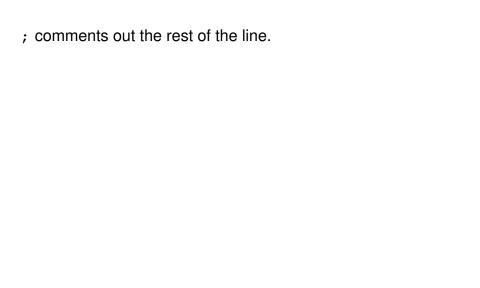


... special reassignable names.

in, *out*, *err*

 \star 1, \star 2, \star 3 are the first, second, and third most recent values.

*e is the most recent exception.



```
... structs
```

```
Define with:
```

```
(defstruct name :field1 :field2 ...)
```

Instantiate with:

```
(struct name val1 val2 ...)
```

Access fields with:

```
(:fieldName structName)
```

A data structure whose parts don't exist until they are accessed.

iterate takes a function ${\tt f}$ and a starting value ${\tt n}$ and produces a lazy infinite series:

```
(n, f(n), f(f(n)), f(f(f(n))), ...)
```

It comes before a vararg, which is available as a list in the

body.

(do exprs*)

Evaluates the expressions in order and returns the value of the last. (for seq-exprs body-expr)

Takes a vector of one or more binding-form/collection-expr pairs, each followed by zero or more modifiers, and yields a lazy sequence of evaluations of expr.

Supported modifiers are: :let [binding-form expr ...], :while test, :when test.

Use partial, followed by a function and fewer than the normal number of arguments.

```
user=> (def equals5 (partial = 5))
#'user/equals5
user=> (equals5 5)
true
```