Wikipedia: a property of some programming languages, in which the primary representation of programs is also a data structure in a primitive type of the language itself.

"code-as-data"

The reader converts the program text into forms which are then converted into Clojure data structures, which are then compiled.

Boolean, number, string, character, keyword, symbol, nil, vector, list, map.

- Integers are automatically promoted to arbitrary-size `BigIntegers` as needed.
- `BigIntegers` and `BigDecimals` have a literal form, a number ending with `M`.

Functions with standard identifiers, operators, Java classes, Clojure namespaces, Java packages, data structures, refs

They may not begin with a number.
They consist of letters, numbers, `+`, `-`, `*`, `/`, `?`.

`.` and _ are also possible, but have special meaning with respect to namespaces.

- They can be multiline.
- They are displayed to the screen with escaped newlines.

They are sequences of characters, so higher-order sequence functions work on them.

It creates a string, much like `toString`, but is n-ary and ignores `nil`.

\backspace, \formfeed, \newline, \return, \space, \tab

```
(apply str [\a \b \c])
```

`apply` is making an n-ary call to `str` given a sequence argument.

Use `true?` and `false?`.

- `(map key)`
- `(keyword-key map)`

Note the second form works only if the key is a keyword.

To document the fact that multiple maps are similar, i.e., they share common keys.

`&` indicates the following param is of variable arity and available as a seq.
`*` indicates the previous param is of variable arity.
`+` indicates the previous param is of variable, non-zero arity.
`?` indicates the previous param is optional.

```
(defstruct name & keys)

(struct name & vals)
```

The keys listed in the struct's definition.

```
(struct-map name & inits)
```

Any missing keys will be given the value `nil` in the resulting struct.

The difference is mostly stylistic, although structs do store their values in indexed slots.

Reader macros are applied prior to the text being broken into forms.

The comment, `;`.

To prevent code becoming unreadable to others, and to prevent Clojure from fragmenting into non-interoperable dialects.

Use the built in predicates `keyword?`, `symbol?`, etc.

```
(defn name doc-string? attr-map? [params*] body)
```

```
(defn name doc-string? attr-map?
 ([params*] body)+ )
```

A macro for:

```
(def name (fn [params*] exprs*))
```

The doc-string and attrs are added to the var metadata.

Just as in the documentation syntax, us a `&` before the final parameter.

- To capture surrounding data in a function created at run-time (i.e., a closure) without giving many different functions the same name.
- To avoid a top-level binding for a function used exclusively inside another one.

```
(# body)
```

where the arguments are given the names `%1`, `%2`, etc. `%` is the same as `%1`.

It's a reader macro.

The initial value of a var it was given in a `def`/`defn` statement.

With the reader macro:

```
#'symbol
```

for `(var symbol)`.

- Store metadata.
- Be rebound differently for each thread.
- Be aliased for unqualified use in other namespaces.

- parameters
- `(let [bindings*] exprs*)`

Destructuring can be used in fn parameter lists, `let`, and macros that expand to either of those.

Vectors can be put in the binding to capture sequential collections, or maps for associative collections.

Use `:as name` in the destructuring. It will bind to the entire collection, not just the part being matched.

It supports `nth`.

`def`, `if`, `do`, `let`, `quote`, `var`, `fn`, `loop`, `recur`, `throw`, `try`, `monitor-enter`, `monitor-exit`.

., new, set!

```
(resolve symbol)
```

... typically the symbol must be quoted to prevent evaluation into whatever it refers to.

`(in-ns name)`

switches to the namespace `name`, creating it if necessary.

They include the package `java.lang`.

`user` also brings in `clojure.core`. Otherwise you would run:

```
(clojure.core/use 'clojure.core)
```

```
(import '(package Class+))
```

```
(use 'clojure.contrib.whatever)
(use '[clojure.contrib.whatever :only (func)])
```

It's available as `*ns*`.

In the namespace declaration.

```
(ns name & references)
```

Each reference an be a `:use` or an `:import`.

It allows you to use multiple expressions where only one is allowed. The last expression is the the result of the whole expression. Previous expressions are executed for side-effects.

It's just the same as `let`, except it introduces a recursion point which `recur` can return to.

```
(loop [bindings *] exprs*)
(recur exprs*)
```

Metadata can be added to collections and symbols.

```
(with-meta object metadata)
```

`=` is like Java's `equals`

`identical?` is like Java's ==

With the `^` macro.

`^x` is like `(meta x)`

```
(assoc map k v & more-kvs)
```

```
(defn #^{:tag BigInt} name [#^{:tag BigInt} i] body)
(defn #^BigInt name [#^BigInt i] body)
(defn name
  ([i] body)
  {:tag BigInt})
```

The reader macro adds metadata directly to a var or parameter.
`with-meta` adds it to the value.