

Clojure Syntax and Data Types



Zachary Bennett

Software Engineer

@z_bennett_ zachbennettcodes.com



Everything is an expression!



Expressions vs. Statements

Statement

**An action to be performed -
does not return a value.**

Expression

**A syntactic unit that can be
evaluated and returns a value.**



`; Basic Clojure syntax`

```
(defn make-name [first-name]  
  (str first-name " Bennett"))
```

```
(make-name "Zachary")
```

```
(println (make-name "Zachary"))
```

```
(def my-number 1)
```

- ◀ **Comments start with a semicolon**
- ◀ **Parentheses define scope and a structure around expressions**
- ◀ **Use the "defn" macro to define a named function. This expression returns the function "make-name"**
- ◀ **An expression which calls the make-name function. This expression evaluates to the string created by the function.**
- ◀ **An expression that produces a side-effect. A "nil" value is returned from this expression.**
- ◀ **Declare a variable named "my-number"**

Numeric and String Literals



String Literals

String

Character

Regular Expression



```
;; String literals
```

```
(def hello "Hello World")
```

```
(def z-char \Z)
```

```
(def my-regexp #"[A-Z]")
```

◀ **The use of double semicolons is a convention used to denote a “header comment”**

◀ **A string literal**

◀ **A character literal**

◀ **A regular expression**

Numeric Literals

Integer

Floating Point

Ratio




```
;; Numeric literals
```

```
(def hundred 100)
```

```
(def my-floating-point 100.12345)
```

```
(def my-ratio 10/7)
```

```
(numerator my-ratio)
```

◀ **An integer**

◀ **A floating point literal**

◀ **A ratio**

◀ **A core, numeric function that returns the integer 10 i.e., the numerator of the given ratio**

Collection Literals



Collection Literals

Sequential

**Ordered collection of
elements**

Hashed

**Unordered collection of
elements**



`:: Sequential Collections`

```
(def my-vector [1 2 3])
```

```
(get my-vector 0)
```

```
(def my-list `(1 2 3))
```

```
(first my-list)
```

◀ **Vectors have indexed elements. New elements are added to the end of the vector.**

◀ **Lists are linked lists. They are not indexed. You have to walk the list to extract the value that you want.**

◀ **Lists are evaluated by invoking the first element as a function – you must quote a list to prevent this and make it a literal list.**

:: Hashed Collections

```
#{"Zachary" "Kalie" "James"}  
(contains? #{"Zachary" "Kalie"} "Kalie")
```

```
{:Zachary 1, :Kalie 8, :James 10}
```

```
{:Zachary 1 :Kalie 8 :James 10}
```

```
(assoc {:Zachary 1} :Brian 12)
```

◀ **A set is created with curly braces preceded by the pound sign. A set is unordered and contains no duplicates.**

◀ **A map is a series of key/value pairs enclosed by curly braces. Remember that commas are optional as they are treated as whitespace!**

◀ **Add a new element to a map**

Maps are used a lot in Clojure!



Symbols and Keywords



Symbol

Symbols are composed of letters, numbers, and other punctuation and are used to refer to something else, like a function, value, namespace, etc.



Symbols are just names!



`:: Symbols`

`(+ 1 2)`

`:: Namespaced Symbols`
`(clojure.core/+ 1 2)`

`:: Special Symbols`

`nil`
`true`
`false`

◀ **Invoke the function that the “+” symbol refers to**

◀ **Use a fully qualified namespaced symbol**

◀ **There are three, special symbols in Clojure which actually refer to core types. These are “nil” (the null value), and the boolean values “true” and “false”.**

Keyword

Keywords are like special symbols that always refer to themselves.



`:: Keywords`

`:foobar`

`::foobar`

`; Keywords are used with maps`

`(def my-map {:Zach 1 :Kalie 2})`

`(get my-map :Kalie)`

`(:Kalie my-map)`

◀ **Keywords are prefaced by a semicolon**

◀ **A namespaced keyword uses two semicolons**

◀ **Keywords are usually used in conjunction with maps to extract values easily. You can think of them as lightweight, constant strings.**

◀ **Keywords can look themselves up!**

Custom Data Types



deftype vs. defrecord

deftype

Programming abstractions

defrecord

Domain-specific abstractions



Protocol

A protocol is a named set of named methods and their signatures, defined using `defprotocol`.



Prefer simple maps unless you are
interacting with protocols or Java!




```
;; deftype / defrecord
```

```
(defrecord Person [name address])
```

```
(deftype Position [x y])
```

- ◀ **defrecord** generates an immutable, persistent map. This generated type has all of the capabilities of a built-in map.
- ◀ **deftype** generates a new type with a constructor, the specified fields, and nothing else. You can also mutate fields in an object generated using **deftype**!

Demo



Create a simple program

Use literal types

- Strings
- Numbers
- Collections
- Symbols

