Clojure Programming Constructs

Ken Q Pu, Faculty of Science, Ontario Tech University

Eval lists

Unless quoted, **Eval** will evaluate a list form as code.

- Function application, or
- Programming constructs

Function application

- Invocation
- Apply

Branch control

- if-else
- case
- cond

Iteration

- for

Block

- do

Function invocation

Invocation of function with arguments

(<fn> <argument> <argument> ...)

```
(+ 1 2 3)
⇒ 6
```

```
((fn [price tax-rate]
  (+ price (* price tax-rate)) 100 0.13)
⇒ 113
```

Invocation with *apply*

(apply <fn> <arguments>)

```
(apply + [1 2 3])
⇒ 6
```

```
(apply (fn [x y z] (+ x (* y z))) [1 2 3])
⇒ 7
```

Branching

Branching

(if <*condition*> <*expression*$_1$> <*expression*$_2$>)

(if <*condition*> <*expression*$_1$>)

Truth in Clojure

What is considered **false**:

- `false`
- `nil`

What is considered **true**:

- Anything that is not considered **false**

Case

```
(case <expression>
    <test value>  <return value>
    <test value> <return value>
    ...
    <default return value>)
```

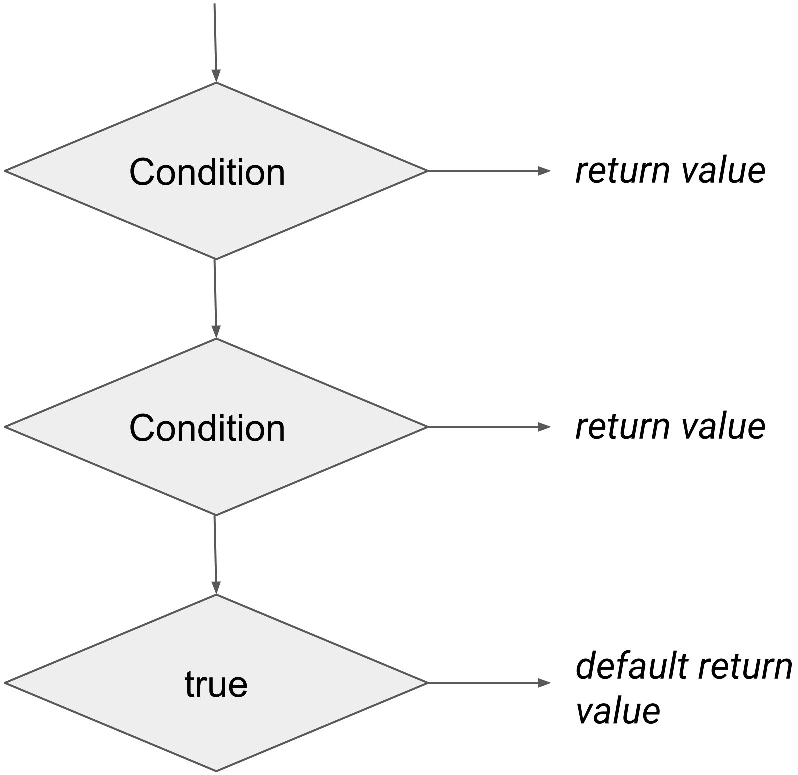
```
(case grade
  :A+ "Great job"
  :A "Very Good job"
  :A- "Good job"
  "Need work")
```

```
(if (= grade :A+)
  "Great job"
  (if (= grade :A)
    "Very Good job"
    (if (= grade :A-)
      "Good job"
      "Need work")))
```

Case construct is designed to reduce code complexity.

Cond

```
(cond
      <condition>  <return value>
      <condition> <return value>
      ...)
```


Condition
return value
Condition
return value
true
default return value

Cond

(cond

<condition> <return value>

<condition> <return value>

...)

```
(cond
    (>= grade 90)   "Great job"
    (>= grade 80)   "Good job"
    true            "Need work")
```

```
(cond
    (>= grade 90)   "Great job"
    (>= grade 80)   "Good job"
    :else           "Need work")
```

Iteration

Iteration with for

Clojure has an extraordinarily rich collection of iterable interfaces to data:

- lists, vectors: iterating over elements
- hashmaps: iterating over key/value pairs
- strings: iterating over characters
- seq: general purpose iterables built on the fly
- ...

Iteration can be done using:

- The for-form
- Functional programming patterns

Basic form of for

(for [*<symbol>* *<iterable>*] *<body>* **)**

Any expression that produces an iterable data.

Basic form of for

```
(for [ <symbol> <iterable> ] <body> )
```

This is a "variable" name that will represent the individual element during the iteration.

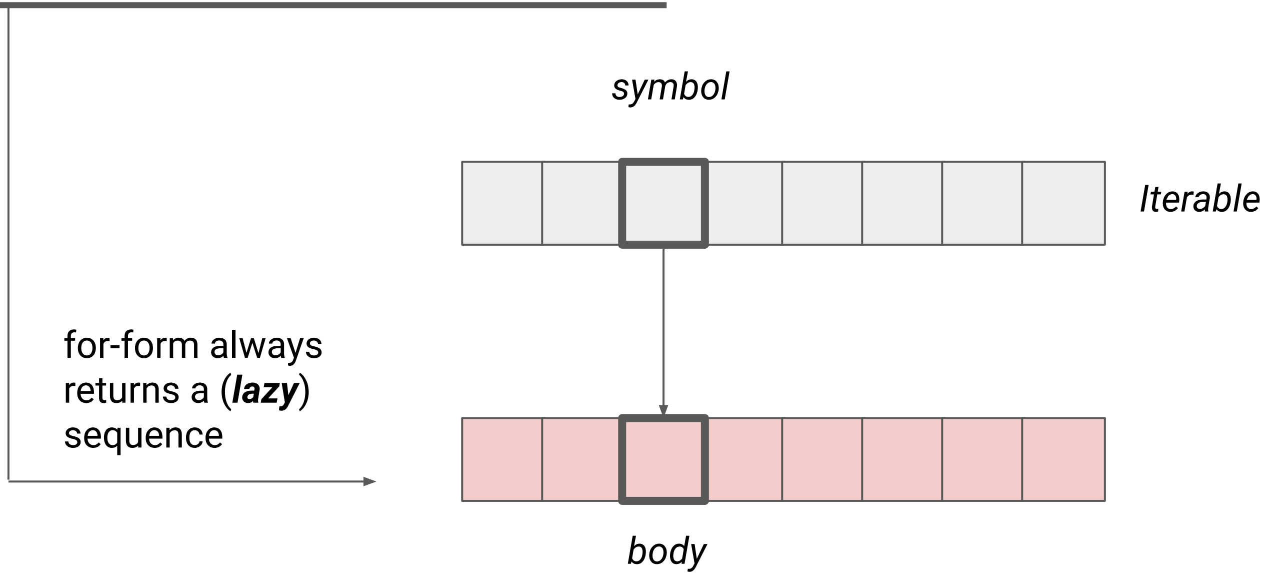
Basic form of for

(for [*<symbol>* *<iterable>*] *<body>* **)**

This expression *computes* the output value based on the data available during the iteration including <*symbol*>.

Basic form of for

(for [*<symbol> <iterable>*] *<body>* **)**



for-form always returns a (***lazy***) sequence

Example

```
(for [i [1 2 3 4 5]] (* 2 i))
⇒ (2 4 6 8 10)
```

```
(for [[name grade] {"Jack" 89
                    "Jill" 90
                    "Joe"  76}]
  (format "%s is %s" name (cond (>= grade 90) "great"
                                (>= 85)       "good"
                                :else         "not great")))
```

Advanced for-forms: nested iteration

```
(for [ <symbol> <iterable>
       <symbol> <iterable>
       ... ]
  <body>)
```

We can have multiple iterables in the for-form. Each iterable will need its own iterator symbol.

This will create *nested* iteration over all the iterables.

Advanced for-forms: additional symbol binding

```
(for [ <x> <iterable>
       <y> <iterable>
       :let [<z> <expression>]]
  <body>)
```

The :let extension allows new symbols created based on the *<expression>*

Advanced for-forms: filtering

```
(for [ <x> <iterable>
       :let [<z> <expression>]
       :when <condition>]
  <body>)
```

The :when expression keeps only elements from the iteration that satisfy the condition.

```
(for [ <x> <iterable>
       :let [<z> <expression>]
       :while <condition>]
  <body>)
```

The :while expression keeps only the initial set of elements that satisfy the condition.

Example

```clojure
(for [[name grade] {"Jack" 89
                    "Jill" 90
                    "Joe"  76}
  :when (>= grade 80)
  :let [status (cond (>= grade 90) "great"
                     (>= 85)       "good"
                     :else         "not great"))]]
  (format "%s is %s" name status)
```

Do blocks

Do form

Lambda Calculus never introduces *side-effects* such as mutating the state of any variables.

However, real-world applications still require side-effects.

Example:

```
(println "Hello world")
```

But many Clojure form only allow **one expression** at specific slots.

Example: **for**-form body, **if**-form condition, etc...

So what if we want to do:

1. evaluate the body, **and**
2. perform side-effect

inside a for-form?

Solution: composite expression using do-form.

Do form

The do-form allows us to package several expressions into a single expression.

The do-form evaluates to the **last expression** in it.

$$\texttt{(do <}\mathit{expression}_1\texttt{> <}\mathit{expression}_2\texttt{> <}\mathit{expression}_3\texttt{> ... <}\mathit{expression}_n\texttt{>)}$$

$$\Rightarrow \texttt{<}\mathit{expression}_n\texttt{>}$$

Example

```
(do (+ 1 2)
    3.14
    (println "Inside do")
    (reverse "Hello"))

Inside do
⇒ "olleH"
```

```
(for [i [1 2 3 4]]
  (do (println "i =", i)
      (* 2 i)))

i = 1
i = 2
i = 3
i = 4
⇒ (2 4 6 8)
```

Lot more to come

- Symbol binding and scoping rules
- Top-level symbol binding
- Recursion
- Tail Recursion