

# Introduction to Functional Programming and Clojure

Jan-Willem van de Meent



# Anatomy of a Clojure Program

```
(ns examples.factorial
  (:gen-class))

(defn factorial
  "computes n * (n-1) * ... * 1"
  [n]
  (if (= n 1)
      1
      (* n (factorial (- n 1)))))

(defn -main
  [& args]
  (doseq [arg args]
    (let [n (Long/parseLong arg)]
      (println "the factorial of" arg
               "is" (factorial n)))))
```

# Anatomy of a Clojure Program

```
(ns examples.factorial  
  (:gen-class))
```

Namespace  
declaration

```
(defn factorial  
  "computes n * (n-1) * ... * 1"  
  [n]  
  (if (= n 1)  
      1  
      (* n (factorial (- n 1)))))
```

```
(defn -main  
  [& args]  
  (doseq [arg args]  
    (let [n (Long/parseLong arg)]  
      (println "the factorial of" arg  
                "is" (factorial n)))))
```

# Anatomy of a Clojure Program

```
(ns examples.factorial  
  (:gen-class))
```

```
(defn factorial  
  "computes n * (n-1) * ... * 1"  
  [n]  
  (if (= n 1)  
      1  
      (* n (factorial (- n 1)))))
```

Recursive  
function

```
(defn -main  
  [& args]  
  (doseq [arg args]  
    (let [n (Long/parseLong arg)]  
      (println "the factorial of" arg  
               "is" (factorial n)))))
```

# Anatomy of a Clojure Program

```
(ns examples.factorial  
  (:gen-class))
```

```
(defn factorial  
  "computes n * (n-1) * ... * 1"  
  [n]  
  (if (= n 1)  
      1  
      (* n (factorial (- n 1)))))
```

```
(defn -main  
  [& args]  
  (doseq [arg args]  
    (let [n (Long/parseLong arg)]  
      (println "the factorial of" arg  
               "is" (factorial n)))))
```

Main  
function

# How do I run this?

```
# get source code for this tutorial
```

```
git clone git@bitbucket.org:probprog/ppaml-summer-school-2016.git  
cd ppaml-summer-school-2016/exercises/
```

```
# option 1: compile to jar and run via java
```

```
lein uberjar
```

```
java -cp target/uberjar/exercises-0.1.0-SNAPSHOT-standalone.jar \  
  examples.factorial 1 2 5 20
```

```
# option 2: run using Leiningen
```

```
lein run -m examples.factorial 1 2 5 20
```

```
# => the factorial of 1 is 1
```

```
# => the factorial of 2 is 2
```

```
# => the factorial of 5 is 120
```

```
# => the factorial of 20 is 2432902008176640000
```

# How do I run this?

*# get source code for this tutorial*

```
git clone git@bitbucket.org:probprog/ppaml-summer-school-2016.git  
cd ppaml-summer-school-2016/exercises/
```

*# option 1: compile to jar and run via java*

```
lein uberjar  
java -cp target/uberjar/exercises-0.1.0-SNAPSHOT-standalone.jar \  
  examples.factorial 1 2 5 20
```

*# option 2: run using Leiningen*

```
lein run -m examples.factorial 1 2 5 20
```

*# => the factorial of 1 is 1*

*# => the factorial of 2 is 2*

*# => the factorial of 5 is 120*

*# => the factorial of 20 is 2432902008176640000*

# How do I run this?

*# get source code for this tutorial*

```
git clone git@bitbucket.org:probprog/ppaml-summer-school-2016.git  
cd ppaml-summer-school-2016/exercises/
```

*# option 1: compile to jar and run via java*

```
lein uberjar
```

```
java -cp target/uberjar/exercises-0.1.0-SNAPSHOT-standalone.jar \  
  examples.factorial 1 2 5 20
```

*# option 2: run using Leiningen*

```
lein run -m examples.factorial 1 2 5 20
```

*# => the factorial of 1 is 1*

*# => the factorial of 2 is 2*

*# => the factorial of 5 is 120*

*# => the factorial of 20 is 2432902008176640000*



# How do I run this?

*# get source code for this tutorial*

```
git clone git@bitbucket.org:probprog/ppaml-summer-school-2016.git  
cd ppaml-summer-school-2016/exercises/
```

*# option 1: compile to jar and run via java*

```
lein uberjar
```

```
java -cp target/uberjar/exercises-0.1.0-SNAPSHOT-standalone.jar \  
  examples.factorial 1 2 5 20
```

*# option 2: run using Leiningen*

```
lein run -m examples.factorial 1 2 5 20
```

```
# => the factorial of 1 is 1
```

```
# => the factorial of 2 is 2
```

```
# => the factorial of 5 is 120
```

```
# => the factorial of 20 is 2432902008176640000
```

# Interactive Shell: the REPL

```
$ lein repl
# => nREPL server started on port 50240 on host
      127.0.0.1 - nrepl://127.0.0.1:50240
# => REPL-y 0.3.7, nREPL 0.2.12
# => Clojure 1.8.0
# => Java HotSpot(TM) 64-Bit Server VM 1.8.0-b132
# =>      Docs: (doc function-name-here)
# =>      (find-doc "part-of-name-here")
# =>      Source: (source function-name-here)
# =>      Javadoc: (javadoc java-object-or-class-here)
# =>      Exit: Control+D or (exit) or (quit)
# =>      Results: Stored in vars *1, *2, *3,
                  an exception in *e
```

```
examples.core=>
```

# Interactive Shell: the REPL


```
examples.core=> (require 'examples.factorial)
;; => nil
```

```
examples.core=> (ns examples.factorial)
;; => #object[clojure.lang.Namespace 0x42cd2abe
"examples.factorial"]
```

```
examples.factorial=> (-main "1" "2" "5" "20")
;; => the factorial of 1 is 1
;; => the factorial of 2 is 2
;; => the factorial of 5 is 120
;; => the factorial of 20 is 2432902008176640000
;; => nil
```

# Gorilla REPL

```
$ lein gorilla
```



The screenshot shows a web browser window titled "Gorilla REPL - exercises" with the address bar displaying "127.0.0.1:62175/worksheet.html". The interface contains two code blocks. The first block contains the code `(ns hello-world (:require [examples.factorial]))` and its output is `nil`. The second block, which is highlighted with a pink border, contains the code `(examples.factorial/-main "1" "2" "5" "10")` and its output is a list of four lines: `the factorial of 1 is 1`, `the factorial of 2 is 2`, `the factorial of 5 is 120`, and `the factorial of 10 is 3628800`. Below the output of the second block is another `nil` output. A hamburger menu icon is visible on the right side of the interface.

```
(ns hello-world
  (:require [examples.factorial]))
```

```
nil
```

```
(examples.factorial/-main "1" "2" "5" "10")
```

```
the factorial of 1 is 1
the factorial of 2 is 2
the factorial of 5 is 120
the factorial of 10 is 3628800
```

```
nil
```

# Anatomy of a Clojure Function

```
(ns examples.factorial
  (:gen-class))

(defn factorial
  "computes n * (n-1) * ... * 1"
  [n]
  (if (= n 1)
      1
      (* n (factorial (- n 1)))))

(defn -main
  [& args]
  (doseq [arg args]
    (let [n (Long/parseLong arg)]
      (println "the factorial of" arg
               "is" (factorial n)))))
```

# Anatomy of a Clojure Function

```
(defn factorial
  "computes n * (n-1) * ... * 1"
  [n]
  (if (= n 1)
      1
      (* n (factorial (- n 1)))))
```

```
def factorial(n):
    '''computes n * (n - 1) * ... * 1'''
    if n == 1:
        return 1
    else:
        return n * factorial(n - 1)
```

# Anatomy of a Clojure Function

```
(defn factorial
  "computes n * (n-1) * ... * 1"
  [n]
  (if (= n 1)
      1
      (* n (factorial (- n 1)))))
```

Name

```
def factorial(n):
    '''computes n * (n - 1) * ... * 1'''
    if n == 1:
        return 1
    else:
        return n * factorial(n - 1)
```

# Anatomy of a Clojure Function

```
(defn factorial
  "computes n * (n-1) * ... * 1"
  [n]
  (if (= n 1)
      1
      (* n (factorial (- n 1)))))
```

Docstring

```
def factorial(n):
    '''computes n * (n - 1) * ... * 1'''
    if n == 1:
        return 1
    else:
        return n * factorial(n - 1)
```



# Anatomy of a Clojure Function

```
(defn factorial
  "computes n * (n-1) * ... * 1"
  [n]
  (if (= n 1)
      1
      (* n (factorial (- n 1)))))
```

Arguments

```
def factorial(n):
    '''computes n * (n - 1) * ... * 1'''
    if n == 1:
        return 1
    else:
        return n * factorial(n - 1)
```

# Anatomy of a Clojure Function

```
(defn factorial
  "computes n * (n-1) * ... * 1"
  [n]
  (if (= n 1)
      1
      (* n (factorial (- n 1)))))
```

Function  
body

```
def factorial(n):
    '''computes n * (n - 1) * ... * 1'''
    if n == 1:
        return 1
    else:
        return n * factorial(n - 1)
```

# Anatomy of a Clojure Function

```
(defn factorial
  "computes n * (n-1) * ... * 1"
  [n]
  (if (= n 1)
      1
      (* n (factorial (- n 1)))))
```

S-expression

```
def factorial(n):
    '''computes n * (n - 1) * ... * 1'''
    if n == 1:
        return 1
    else:
        return n * factorial(n - 1)
```

Block  
statement

# Anatomy of an Expression

```
(defn factorial  
  "computes n * (n-1) * ... * 1"  
  [n]  
  (if (= n 1)  
      1  
      (* n (factorial (- n 1)))))
```

# Anatomy of an Expression

```
(defn factorial  
  "computes n * (n-1) * ... * 1"  
  [n]  
  (if (= n 1)  
      1  
      (* n (factorial (- n 1)))))
```

# Anatomy of an Expression

```
(defn factorial
  "computes n * (n-1) * ... * 1"
  [n]
  (if (= n 1)
      1
      (* n (factorial (- n 1)))))
```

# Anatomy of an Expression

```
(defn factorial  
  "computes n * (n-1) * ... * 1"  
  [n]  
  (if (= n 1)  
      1  
      (* n (factorial (- n 1)))))
```

# Anatomy of an Expression

```
(defn factorial
  "computes n * (n-1) * ... * 1"
  [n]
  (if (= n 1)
      1
      (* n (factorial (- n 1)))))
```



# Anatomy of an Expression

```
(defn factorial
  "computes n * (n-1) * ... * 1"
  [n]
  (if (= n 1)
      1
      (* n (factorial (- n 1)))))
```

# Anatomy of an Expression

```
(defn factorial  
  "computes n * (n-1) * ... * 1"  
  [n]  
  (if (= n 1)  
      1  
      (* n (factorial (- n 1)))))
```

# Anatomy of an Expression

```
(defn factorial
  "computes n * (n-1) * ... * 1"
  [n]
  (if (= n 1)
      1
      (* n (factorial (- n 1)))))
```

# Anatomy of an Expression

```
(defn factorial  
  "computes n * (n-1) * ... * 1"  
  [n]  
  (if (= n 1)  
      1  
      (* n (factorial (- n 1)))))
```

# Anatomy of an Expression

```
(defn factorial
  "computes n * (n-1) * ... * 1"
  [n]
  (if (= n 1)
      1
      (* n (factorial (- n 1)))))
```

# Anatomy of an Expression

```
(defn factorial
  "computes n * (n-1) * ... * 1"
  [n]
  (if (= n 1)
      1
      (* n (factorial (- n 1)))))
```

expression ::= symbol | literal | (operator ...)

# Anatomy of an Expression

```
(defn factorial  
  "computes n * (n-1) * ... * 1"  
  [n]  
  (if (= n 1)  
      1  
      (* n (factorial (- n 1)))))
```

expression ::= symbol | literal | (operator ...)

# Anatomy of an Expression

```
(defn factorial  
  "computes n * (n-1) * ... * 1"  
  [n]  
  (if (= n 1)  
      1  
      (* n (factorial (- n 1)))))
```

expression ::= symbol | literal | (operator ...)



# Anatomy of an Expression

```
(defn factorial
  "computes n * (n-1) * ... * 1"
  [n]
  (if (= n 1)
      1
      (* n (factorial (- n 1)))))
```

expression ::= symbol | literal | (operator ...)

# Anatomy of an Expression

```
(defn factorial
  "computes n * (n-1) * ... * 1"
  [n]
  (if (= n 1)
      1
      (* n (factorial (- n 1)))))
```

expression ::= symbol | literal | (operator ...)

operator ::= special | function | macro

# Anatomy of an Expression

```
(defn factorial
  "computes n * (n-1) * ... * 1"
  [n]
  (if (= n 1)
    1
    (* n (factorial (- n 1)))))
```

expression ::= symbol | literal | (operator ...)

operator ::= special | function | macro

# Anatomy of an Expression

```
(defn factorial
  "computes n * (n-1) * ... * 1"
  [n]
  (if (= n 1)
      1
      (* n (factorial (- n 1)))))
```

expression ::= symbol | literal | (operator ...)

operator ::= special | function | macro

# Anatomy of an Expression

```
(defn factorial
  "computes n * (n-1) * ... * 1"
  [n]
  (if (= n 1)
      1
      (* n (factorial (- n 1)))))
```

expression ::= symbol | literal | (operator ...)

operator ::= special | function | macro

# Anatomy of an Expression

```
(defn factorial
  "computes n * (n-1) * ... * 1"
  [n]
  (if (= n 1)
    1
    (* n (factorial (- n 1)))))
```

expression ::= symbol | literal | (operator ...)

operator ::= special | function | macro

special ::= def | if | fn | let | loop | recur |  
do | new | . | throw | set! | quote | var

# Anatomy of an Expression

```
(defn factorial
  "computes n * (n-1) * ... * 1"
  [n]
  (if (= n 1)
      1
      (* n (factorial (- n 1)))))
```

expression ::= symbol | literal | (operator ...)

operator ::= special | function | macro

special ::= def | if | fn | let | loop | recur |  
do | new | . | throw | set! | quote | var

# Data Types

## Atomic

*;; symbols*  
(**symbol** "ada"), ada

*;; keywords*  
**:ada**

*;; integers, doubles, ratios*  
1234, 1.234, 12/34

*;; strings, characters*  
"ada", \a \d \a

*;; booleans, null*  
true, false, nil

*;; regular expressions*  
**#"a\*b"**

## Collections

*;; lists*  
(**list** 1 2 3), (1 2 3)

*;; hash maps*  
{**:a** 1 **:b** 2}

*;; vectors*  
[1 2 3]

*;; sets*  
**#**{1 2 3}

*;; everything nests*  
{**:a** [[1 2] [3 4]]  
  **:b** #{5 6 (**list** 7 8)}  
  **:c** {"d" 9 \e 10}}



# Data Types

## Atomic

*;; symbols*  
(symbol "ada"), ada

*;; keywords*  
:ada

*;; integers, doubles, ratios*  
1234, 1.234, 12/34

*;; strings, characters*  
"ada", \a \d \a

*;; booleans, null*  
true, false, nil

*;; regular expressions*  
#"a\*b"

## Collections

*;; lists*  
(list 1 2 3), (1 2 3)

*;; hash maps*  
{:a 1 :b 2}

*;; vectors*  
[1 2 3]

*;; sets*  
#{1 2 3}

*;; everything nests*  
{:a [[1 2] [3 4]]  
:b #{5 6 (list 7 8)}  
:c {"d" 9 \e 10}}

# Data Types

## Atomic

*;; symbols*  
(symbol "ada"), ada

*;; keywords*  
:ada

*;; integers, doubles, ratios*  
1234, 1.234, 12/34

*;; strings, characters*  
"ada", \a \d \a

*;; booleans, null*  
true, false, nil

*;; regular expressions*  
#"a\*b"

## Collections

*;; lists*  
(list 1 2 3), (1 2 3)

*;; hash maps*  
{:a 1 :b 2}

*;; vectors*  
[1 2 3]

*;; sets*  
#{1 2 3}

*;; everything nests*  
{:a [[1 2] [3 4]]  
:b #{5 6 (list 7 8)}  
:c {"d" 9 \e 10}}

# Data Types

## Atomic

*;; symbols*  
(symbol "ada"), ada

*;; keywords*  
:ada

*;; integers, doubles, ratios*  
1234, 1.234, 12/34

*;; strings, characters*  
"ada", \a \d \a

*;; booleans, null*  
true, false, nil

*;; regular expressions*  
#"a\*b"

## Collections

*;; lists*  
(list 1 2 3), (1 2 3)

*;; hash maps*  
{:a 1 :b 2}

*;; vectors*  
[1 2 3]

*;; sets*  
#{1 2 3}

*;; everything nests*  
{:a [[1 2] [3 4]]  
:b #{5 6 (list 7 8)}  
:c {"d" 9 \e 10}}

# Data Types

## Atomic

*;; symbols*  
(symbol "ada"), ada

*;; keywords*  
:ada

*;; integers, doubles, ratios*  
1234, 1.234, 12/34

*;; strings, characters*  
"ada", \a \d \a

*;; booleans, null*  
true, false, nil

*;; regular expressions*  
#"a\*b"

## Collections

*;; lists*  
(list 1 2 3), (1 2 3)

*;; hash maps*  
{:a 1 :b 2}

*;; vectors*  
[1 2 3]

*;; sets*  
#{1 2 3}

*;; everything nests*  
{:a [[1 2] [3 4]]  
:b #{5 6 (list 7 8)}  
:c {"d" 9 \e 10}}

# Data Types

## Atomic

*;; symbols*  
(symbol "ada"), ada

*;; keywords*  
:ada

*;; integers, doubles, ratios*  
1234, 1.234, 12/34

*;; strings, characters*  
"ada", \a \d \a

*;; booleans, null*  
true, false, nil

*;; regular expressions*  
#"a\*b"

## Collections

*;; lists*  
(list 1 2 3), (1 2 3)

*;; hash maps*  
{:a 1 :b 2}

*;; vectors*  
[1 2 3]

*;; sets*  
#{1 2 3}

*;; everything nests*  
{:a [[1 2] [3 4]]  
:b #{5 6 (list 7 8)}  
:c {"d" 9 \e 10}}

# Data Types

## Atomic

*;; symbols*  
(symbol "ada"), ada

*;; keywords*  
:ada

*;; integers, doubles, ratios*  
1234, 1.234, 12/34

*;; strings, characters*  
"ada", \a \d \a

*;; booleans, null*  
true, false, nil

*;; regular expressions*  
#"a\*b"

## Collections

*;; lists*  
(list 1 2 3), (1 2 3)

*;; hash maps*  
{:a 1 :b 2}

*;; vectors*  
[1 2 3]

*;; sets*  
#{1 2 3}

*;; everything nests*  
{:a [[1 2] [3 4]]  
:b #{5 6 (list 7 8)}  
:c {"d" 9 \e 10}}

# Data Types

## Atomic

*;; symbols*  
(symbol "ada"), ada

*;; keywords*  
:ada

*;; integers, doubles, ratios*  
1234, 1.234, 12/34

*;; strings, characters*  
"ada", \a \d \a

*;; booleans, null*  
true, false, nil

*;; regular expressions*  
#"a\*b"

## Collections

*;; lists*  
(list 1 2 3), (1 2 3)

*;; hash maps*  
{"a" 1 "b" 2}

*;; vectors*  
[1 2 3]

*;; sets*  
#{1 2 3}

*;; everything nests*  
{:a [[1 2] [3 4]]  
 :b #{5 6 (list 7 8)}  
 :c {"d" 9 \e 10}}

# Data Types

## Atomic

*;; symbols*  
(symbol "ada"), ada

*;; keywords*  
:ada

*;; integers, doubles, ratios*  
1234, 1.234, 12/34

*;; strings, characters*  
"ada", \a \d \a

*;; booleans, null*  
true, false, nil

*;; regular expressions*  
#"a\*b"

## Collections

*;; lists*  
(list 1 2 3), (1 2 3)

*;; hash maps*  
{:a 1 :b 2}

*;; vectors*  
[1 2 3]

*;; sets*  
#{1 2 3}

*;; everything nests*  
{:a [[1 2] [3 4]]  
:b #{5 6 (list 7 8)}  
:c {"d" 9 \e 10}}



# Data Types

## Atomic

*;; symbols*  
(symbol "ada"), ada

*;; keywords*  
:ada

*;; integers, doubles, ratios*  
1234, 1.234, 12/34

*;; strings, characters*  
"ada", \a \d \a

*;; booleans, null*  
true, false, nil

*;; regular expressions*  
#"a\*b"

## Collections

*;; lists*  
(list 1 2 3), (1 2 3)

*;; hash maps*  
{:a 1 :b 2}

*;; vectors*  
[1 2 3]

*;; sets*  
#{1 2 3}

*;; everything nests*  
{:a [[1 2] [3 4]]  
:b #{5 6 (list 7 8)}  
:c {"d" 9 \e 10}}

# Data Types

## Atomic

*;; symbols*  
(symbol "ada"), ada

*;; keywords*  
:ada

*;; integers, doubles, ratios*  
1234, 1.234, 12/34

*;; strings, characters*  
"ada", \a \d \a

*;; booleans, null*  
true, false, nil

*;; regular expressions*  
#"a\*b"

## Collections

*;; lists*  
(list 1 2 3), (1 2 3)

*;; hash maps*  
{:a 1 :b 2}

*;; vectors*  
[1 2 3]

*;; sets*  
#{1 2 3}

*;; everything nests*  
{:a [[1 2] [3 4]]  
:b #{5 6 (list 7 8)}  
:c {"d" 9 \e 10}}

# Data Types

## Atomic

*;; symbols*  
(**symbol** "ada"), **ada**

*;; keywords*  
**:ada**

*;; integers, doubles, ratios*  
1234, 1.234, 12/34

*;; strings, characters*  
"ada", \a \d \a

*;; booleans, null*  
true, false, nil

*;; regular expressions*  
**#"a\*b"**

## Collections

*;; lists*  
(**list** 1 2 3), **(1 2 3)**

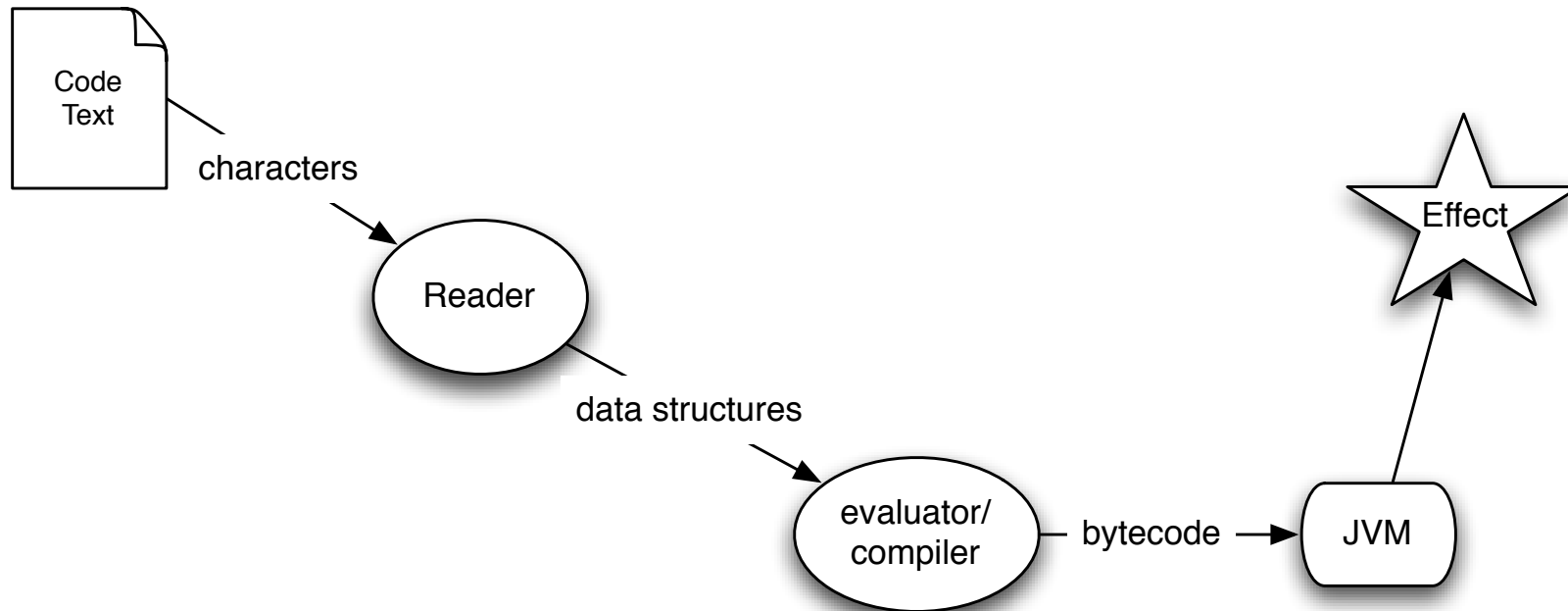
*;; hash maps*  
**{:a 1 :b 2}**

*;; vectors*  
**[1 2 3]**

*;; sets*  
**#{1 2 3}**

*;; everything nests*  
**{:a [[1 2] [3 4]]**  
  **:b #{5 6 (list 7 8)}**  
  **:c {"d" 9 \e 10}}**

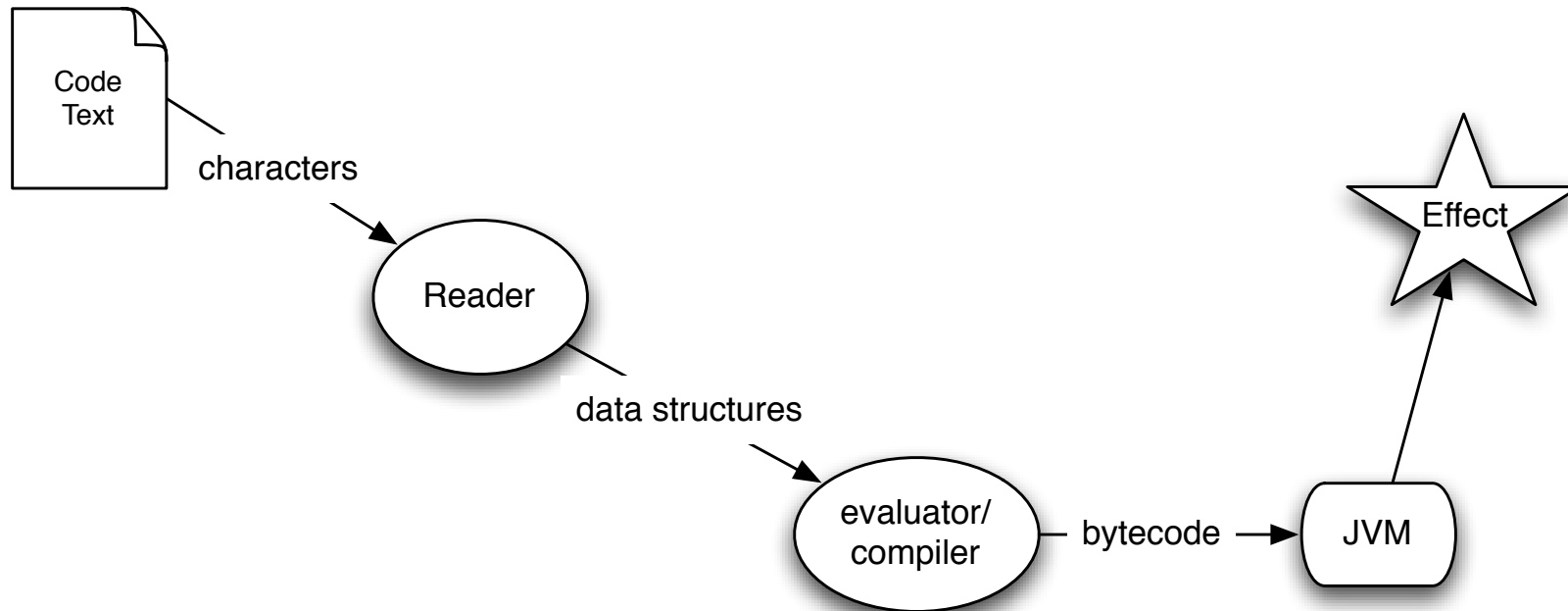
# Evaluation in Clojure



```
(let [expr (read-string "(+ 1 2)")]  
  (prn expr) ; => (+ 1 2)  
  (prn (class expr)) ; => clojure.lang.PersistentList  
  (prn (class (first expr))) ; => clojure.lang.Symbol  
  (eval expr)) ; => 3
```

(image credit: Rich Hickey)

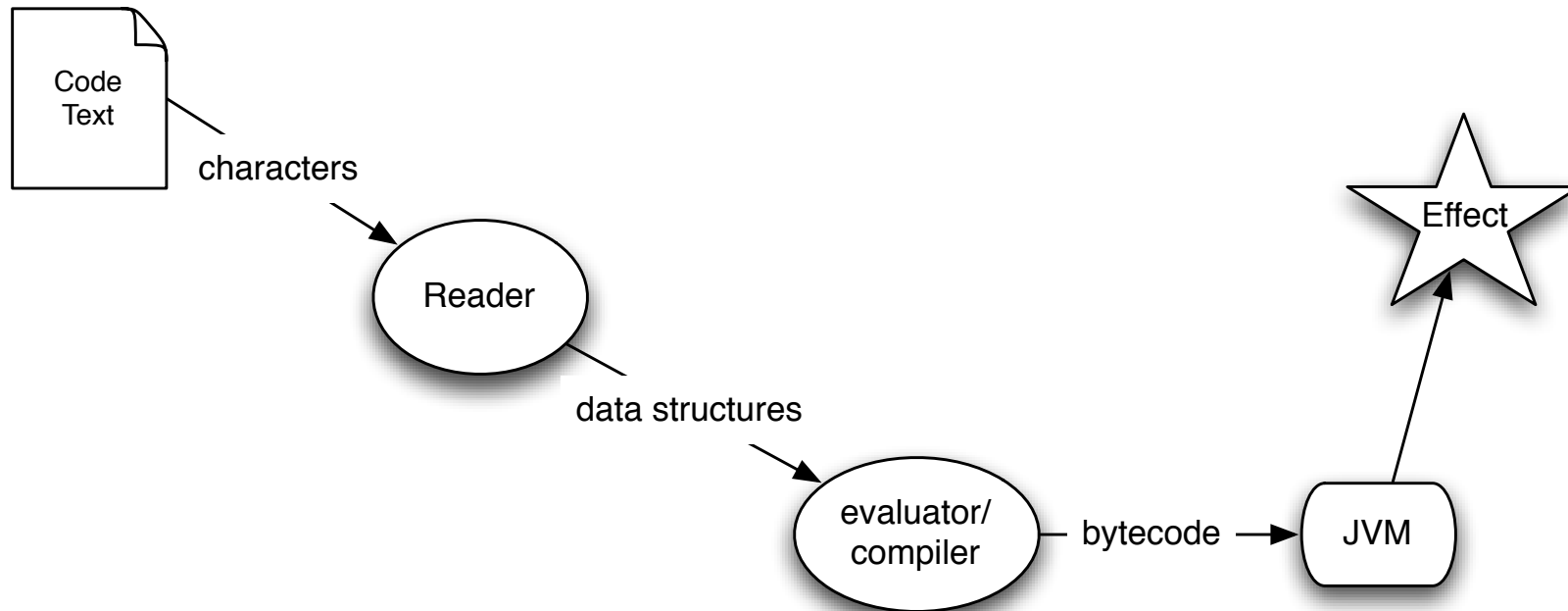
# Evaluation in Clojure



```
(let [expr (read-string "(+ 1 2)")]  
  (prn expr) ; => (+ 1 2)  
  (prn (class expr)) ; => clojure.lang.PersistentList  
  (prn (class (first expr))) ; => clojure.lang.Symbol  
  (eval expr)) ; => 3
```

(image credit: Rich Hickey)

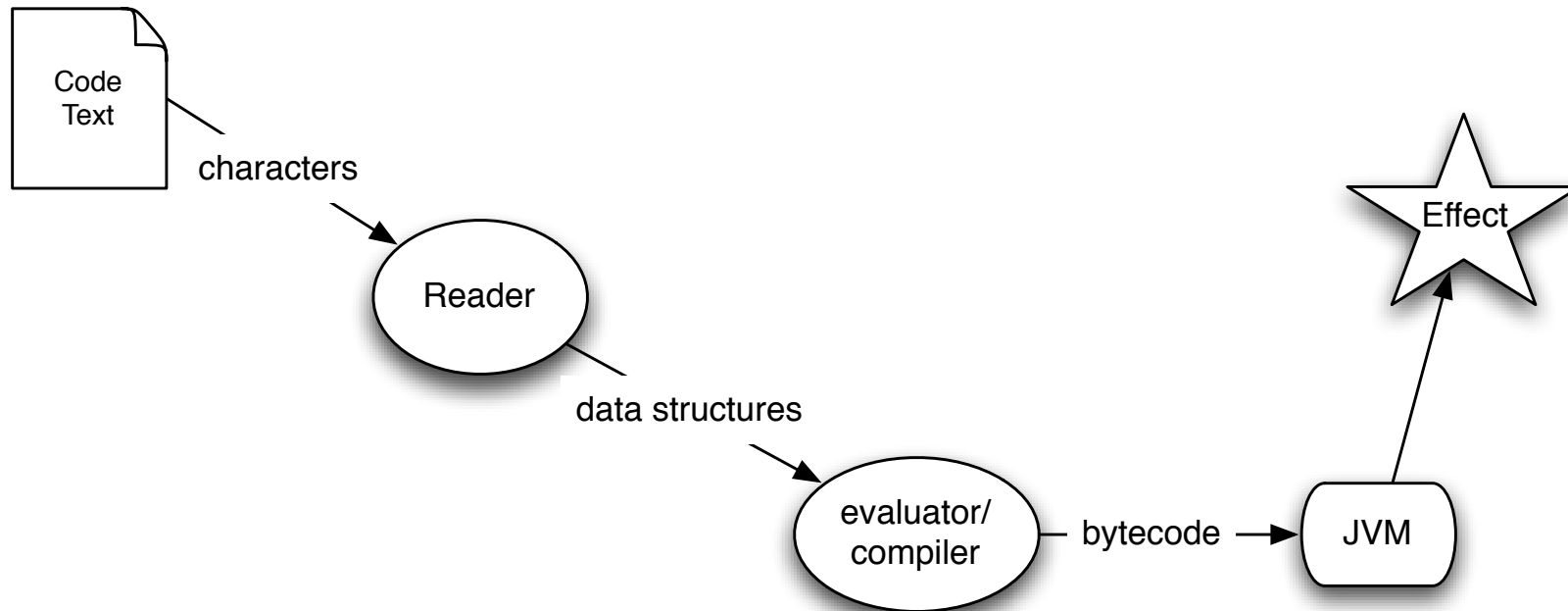
# Evaluation in Clojure



```
(let [expr (read-string "(+ 1 2)")]  
  (prn expr) ; => (+ 1 2)  
  (prn (class expr)) ; => clojure.lang.PersistentList  
  (prn (class (first expr))) ; => clojure.lang.Symbol  
  (eval expr)) ; => 3
```

(image credit: Rich Hickey)

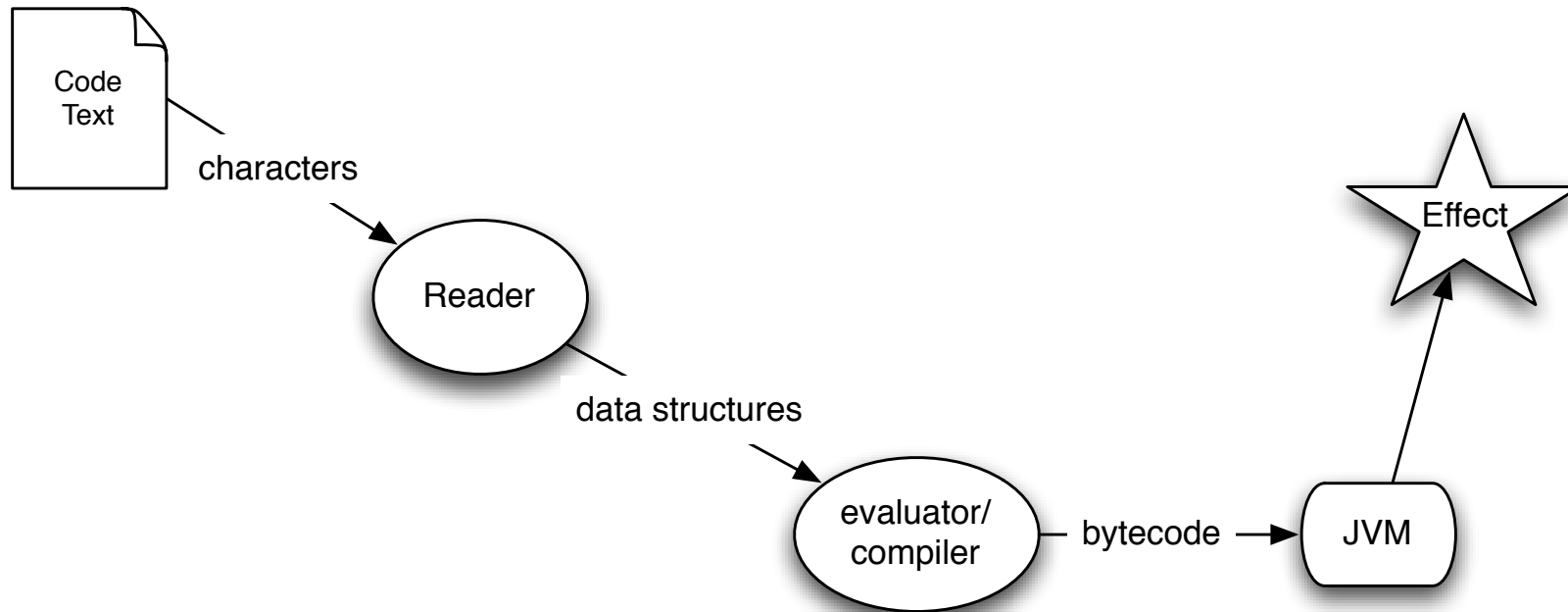
# Evaluation in Clojure



```
(let [expr (read-string "(+ 1 2)")]  
  (prn expr) ; => (+ 1 2)  
  (prn (class expr)) ; => clojure.lang.PersistentList  
  (prn (class (first expr))) ; => clojure.lang.Symbol  
  (eval expr)) ; => 3
```

(image credit: Rich Hickey)

# Evaluation in Clojure

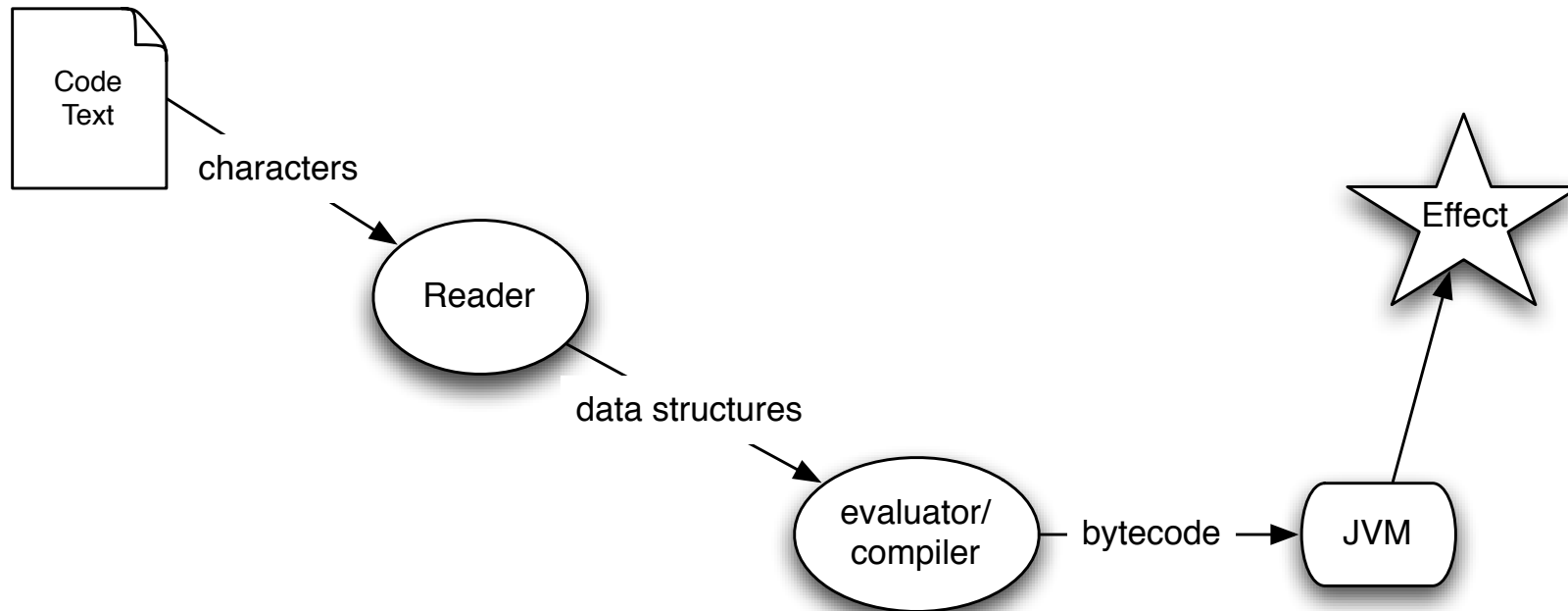


```
(let [expr (read-string "(+ 1 2)")]  
  (prn expr) ; => (+ 1 2)  
  (prn (class expr)) ; => clojure.lang.PersistentList  
  (prn (class (first expr))) ; => clojure.lang.Symbol  
  (eval expr)) ; => 3
```

(image credit: Rich Hickey)



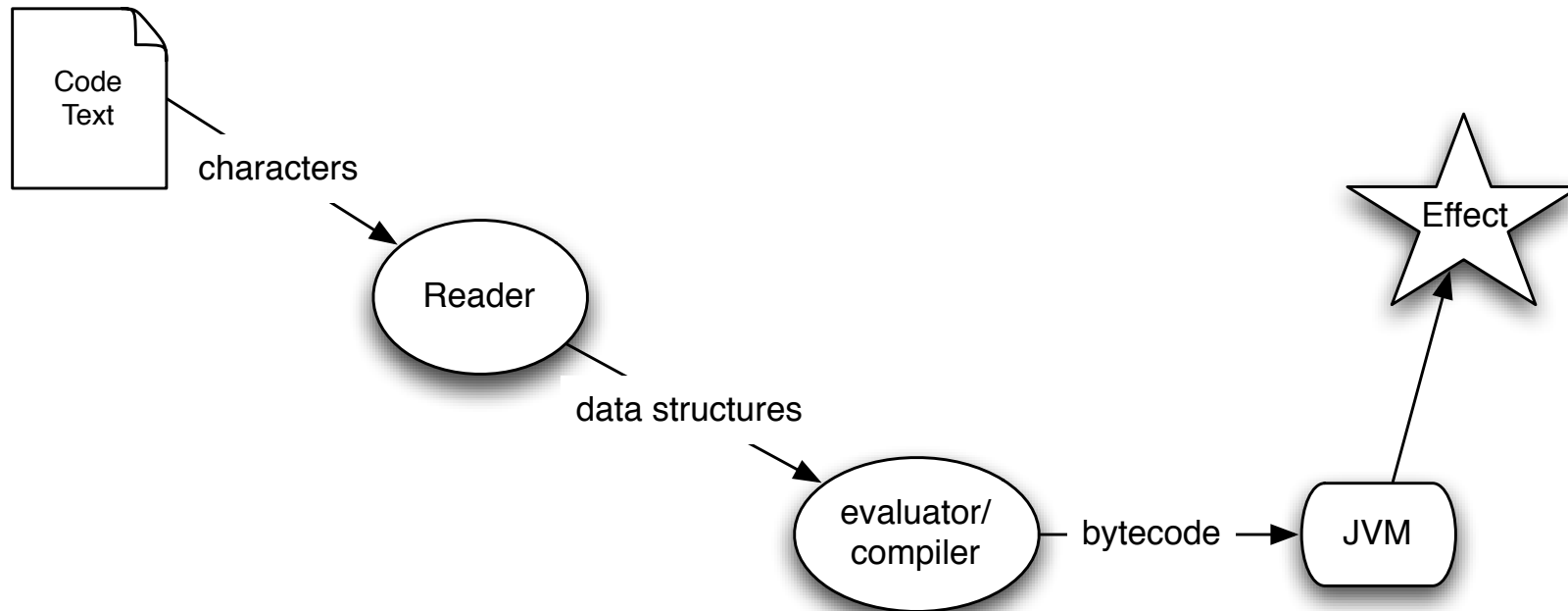
# Evaluation in Clojure



```
(let [expr (read-string "(+ 1 2)")]  
  (prn expr) ; => (+ 1 2)  
  (prn (class expr)) ; => clojure.lang.PersistentList  
  (prn (class (first expr))) ; => clojure.lang.Symbol  
  (eval expr)) ; => 3
```

(image credit: Rich Hickey)

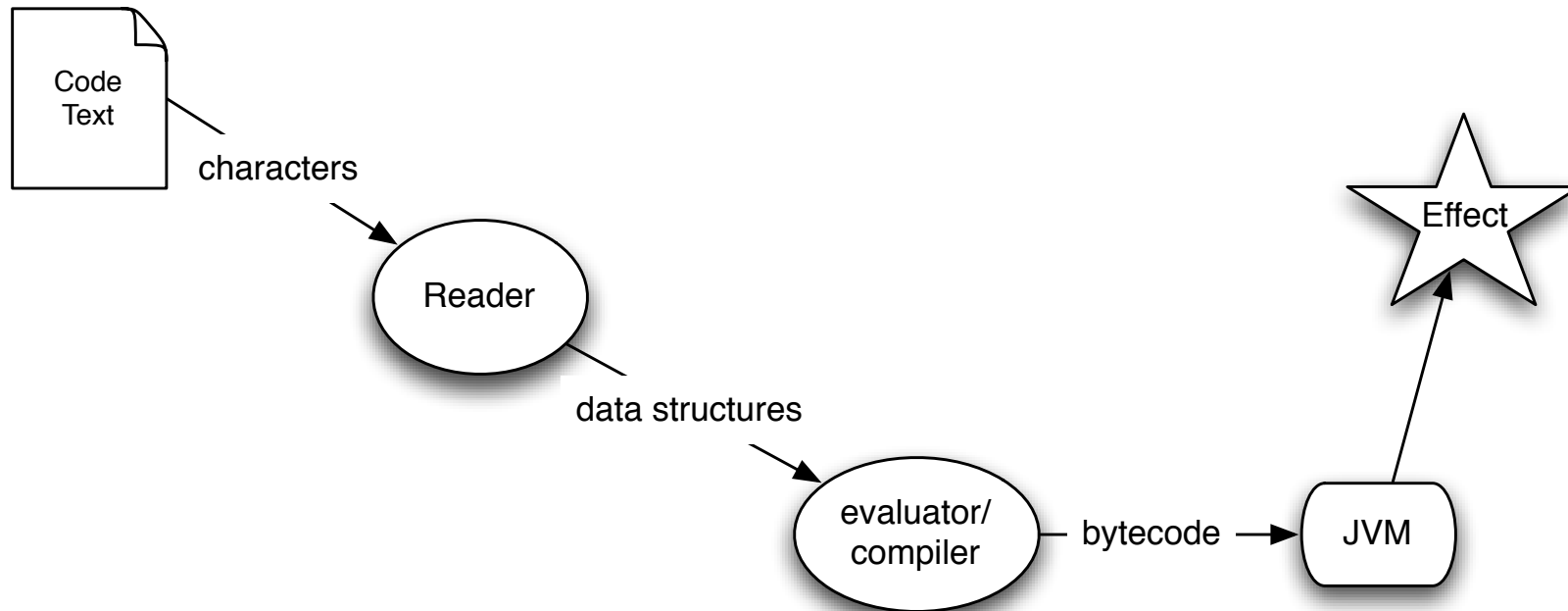
# Evaluation in Clojure



```
(let [expr (read-string "(+ 1 2)")]  
  (prn expr) ; => (+ 1 2)  
  (prn (class expr)) ; => clojure.lang.PersistentList  
  (prn (class (first expr))) ; => clojure.lang.Symbol  
  (eval expr)) ; => 3
```

(image credit: Rich Hickey)

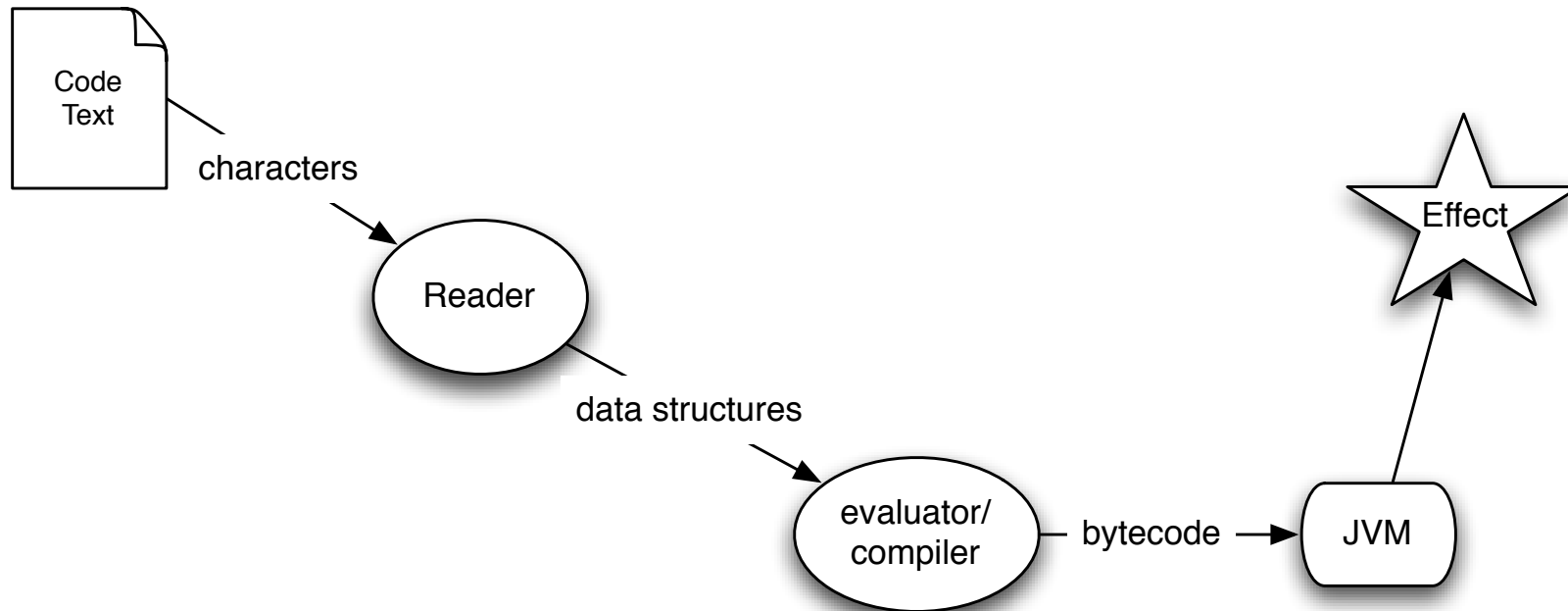
# Evaluation in Clojure



```
(let [expr (quote (+ 1 2))]  
  (prn expr) ; => (+ 1 2)  
  (prn (class expr)) ; => clojure.lang.PersistentList  
  (prn (class (first expr))) ; => clojure.lang.Symbol  
  (eval expr)) ; => 6
```

(image credit: Rich Hickey)

# Evaluation in Clojure



```
(let [expr '(+ 1 2)]  
  (prn expr) ; => (+ 1 2)  
  (prn (class expr)) ; => clojure.lang.PersistentList  
  (prn (class (first expr))) ; => clojure.lang.Symbol  
  (eval expr)) ; => 6
```

(image credit: Rich Hickey)

# Macros

```
(defmacro unless  
  "Inverted 'if"  
  [pred then else]  
  (list 'if pred else then))
```

```
(def flavor :tasty)
```

```
(unless (= flavor :tasty)  
  :yuk  
  :yum)
```

; ~> *(macro-expansion)*

```
(if (= flavor :tasty)  
  :yum  
  :yuk)
```

; => *(evaluation)*

```
:yum
```

# Macros

```
(defmacro dbg
  "Prints an expression and
  its value for debugging."
  [expr]
  `(let [value# ~expr]
      (println "[dbg]"
               '~expr
               value#)
      value#))
```

```
(dbg (+ 1 2))
; => [dbg] (+ 1 2) 3
; => 3

(macroexpand '(dbg (+ 1 2)))
; => (let* [value__23707__auto__
            (+ 1 2)]
      (clojure.core/println
       "[dbg]"
       (quote (+ 1 2))
       value__23707__auto__
       value__23707__auto__))
```

# Method Dispatch

## Collections

```
(count (list 1 2 3))
```

```
; => 3
```

```
(count [1 2 3 4 5])
```

```
; => 5
```

```
(count {:a 1, :b 2})
```

```
; => 2
```

```
(conj (list 1 2 3) 0)
```

```
; => (0 1 2 3)
```

```
(conj [1 2 3] 4)
```

```
; => [1 2 3 4]
```

```
(conj {:a 1, :b 2} [:c 3])
```

```
; => {:a 1 :b 2 :c 3}
```

## Sequences

```
(seq [1 2 3])
```

```
; => (1 2 3)
```

```
(seq {:a 1, :b 2})
```

```
; => ([:a 1] [:b 2])
```

```
(seq (list))
```

```
; => nil
```

```
(first [1 2 3])
```

```
; => 1
```

```
(rest [1 2 3])
```

```
; => (2 3)
```

```
(cons 0 [1 2 3])
```

```
; => (0 1 2 3)
```

# Method Dispatch

## Associative

*(vectors, maps)*

```
(get {:a 1 :b 2} :a)  
; => 1  
(get ["a" "b"] 0)  
; => "a"
```

```
(assoc {:a 1 :b 2} :c 3)  
; => {:a 1 :b 2 :c 3}  
(assoc ["a" "b"] 2 "c")  
; => ["a" "b" "c"]
```

## Stacks

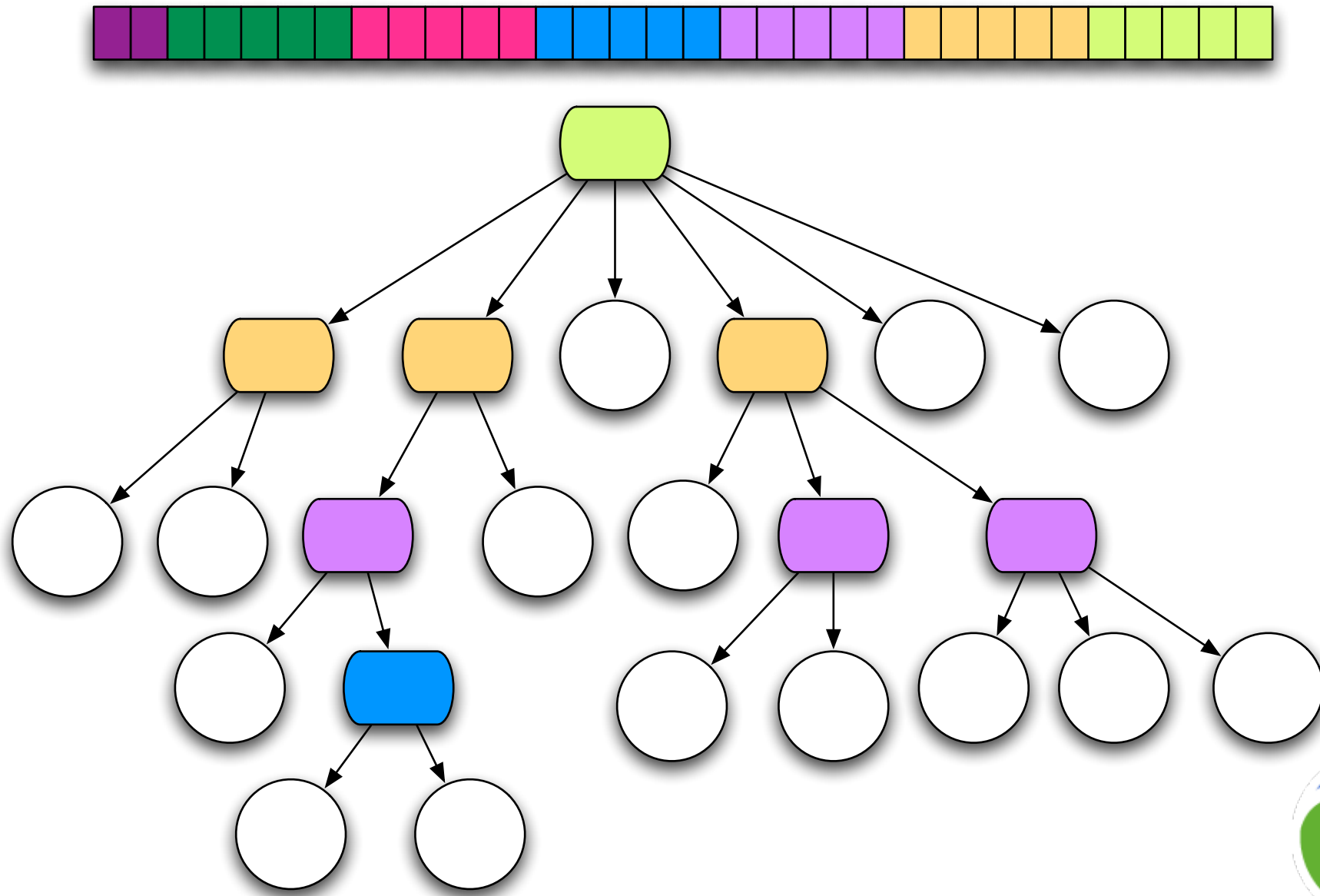
*(lists, vectors)*

```
(peek (list 1 2 3))  
; => 1  
(peek [1 2 3])  
; => 3
```

```
(pop (list 1 2 3))  
; => (list 2 3)  
(pop [1 2 3])  
; => [1 2]
```

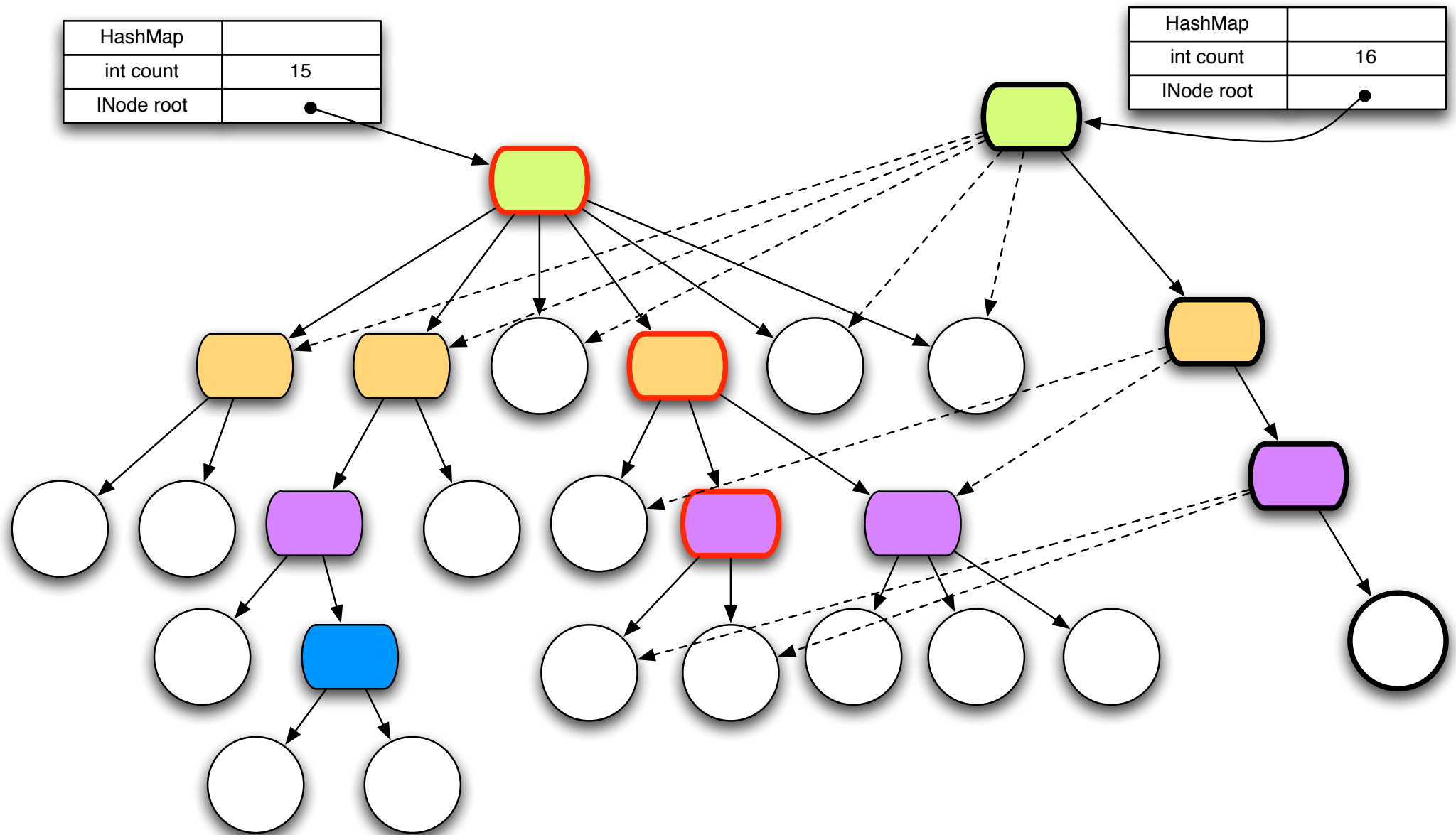


# Bit-partitioned Hash Tries



(image credit: Rich Hickey)

# Path Copying



(image credit: Rich Hickey)

# Looping

```
(defn factorial
  "computes n * (n-1) * ... * 1"
  [n]
  (if (= n 1)
      1
      (* n (factorial (- n 1)))))
```

# Looping

```
(defn factorial
  "computes n * (n-1) * ... * 1"
  [n]
  (if (= n 1)
      1
      (* n (factorial (- n 1)))))
```

```
(factorial 21)
; => ArithmeticException integer overflow
;    clojure.lang.Numbers.throwIntOverflow (Numbers.java:1501)
```

# Looping

```
(defn factorial
  "computes n * (n-1) * ... * 1"
  [n]
  (if (= n 1)
    1N
    (* n (factorial (- n 1)))))
```

```
(factorial 21)
; => ArithmeticException integer overflow
;   clojure.lang.Numbers.throwIntOverflow (Numbers.java:1501)
```

# Looping

```
(defn factorial
  "computes n * (n-1) * ... * 1"
  [n]
  (if (= n 1)
      1
      (* n (factorial (- n 1)))))
```

```
(factorial 21)
; => ArithmeticException integer overflow
;   clojure.lang.Numbers.throwIntOverflow (Numbers.java:1501)
```

# Looping

```
(defn factorial
  "computes n * (n-1) * ... * 1"
  [n]
  (if (= n 1)
    1
    (*' n (factorial (- n 1)))))
```

```
(factorial 10000)
; => StackOverflowError
      clojure.lang.Numbers.equal (Numbers.java:216)
```

# Looping

```
(defn factorial
  "computes n * (n-1) * ... * 1"
  [n]
  (if (= n 1)
      1
      (* n (factorial (- n 1)))))
```

```
def factorial(n):
    '''computes n * (n - 1) * ... * 1'''
    if n == 1:
        return 1
    else:
        return n * factorial(n - 1)
```



# Looping

```
(defn factorial
  "computes n * (n-1) * ... * 1"
  [n]
  (if (= n 1)
      1
      (* n (factorial (- n 1)))))
```

```
def factorial(n):
    '''computes n * (n - 1) * ... * 1'''
    result = 1
    for i in range(2, n + 1):
        result *= i
    return result
```

# Looping

```
(defn factorial
  "computes n * (n-1) * ... * 1"
  [n]
  (if (= n 1)
    1
    (* n (factorial (- n 1)))))
```

```
def factorial(n):
    '''computes n * (n - 1) * ... * 1'''
    result = 1
    ival = range(2, n + 1)
    while ival:
        i = ival.pop(0)
        result *= i
    return result
```

# Looping

```
(defn factorial [n]
  "computes n * (n-1) * ... * 1"
  (loop [result 1
        ival (range 2 (+ n 1))]
    (if (seq ival)
      (recur (* result (first ival))
             (rest ival))
      result)))
```

```
def factorial(n):
    '''computes n * (n - 1) * ... * 1'''
    result = 1
    ival = range(2, n + 1)
    while ival:
        i = ival.pop(0)
        result *= i
    return result
```

# Looping

```
(defn factorial [n]
  "computes n * (n-1) * ... * 1"
  (loop [result 1
        ival (range 2 (+ n 1))]
    (if (seq ival)
      (recur (* result (first ival))
             (rest ival))
      result)))
```

```
def factorial(n):
    '''computes n * (n - 1) * ... * 1'''
    result = 1
    ival = range(2, n + 1)
    while ival:
        i = ival.pop(0)
        result *= i
    return result
```

# Looping

```
(defn factorial [n]
  "computes n * (n-1) * ... * 1"
  (loop [result 1
        ival (range 2 (+ n 1))]
    (if (seq ival)
        (recur (* result (first ival))
               (rest ival))
        result)))
```

Initial values

```
def factorial(n):
    '''computes n * (n - 1) * ... * 1'''
    result = 1
    ival = range(2, n + 1)
    while ival:
        i = ival.pop(0)
        result *= i
    return result
```

# Looping

```
(defn factorial [n]
  "computes n * (n-1) * ... * 1"
  (loop [result 1
        ival (range 2 (+ n 1))]
    (if (seq ival)
        (recur (* result (first ival))
               (rest ival))
        result)))
```

Any values  
for `i` remaining?

```
def factorial(n):
    '''computes n * (n - 1) * ... * 1'''
    result = 1
    ival = range(2, n + 1)
    while ival:
        i = ival.pop(0)
        result *= i
    return result
```

# Looping

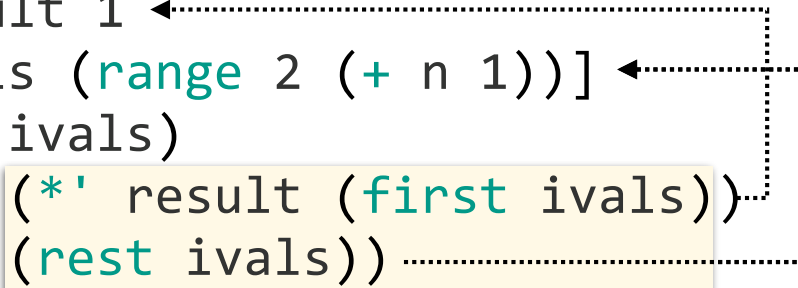
```
(defn factorial [n]
  "computes n * (n-1) * ... * 1"
  (loop [result 1
        ival (range 2 (+ n 1))]
    (if (seq ival)
      (recur (* result (first ival))
             (rest ival))
      result)))
```

Compute values  
for next iteration

```
def factorial(n):
    '''computes n * (n - 1) * ... * 1'''
    result = 1
    ival = range(2, n + 1)
    while ival:
        i = ival.pop(0)
        result *= i
    return result
```

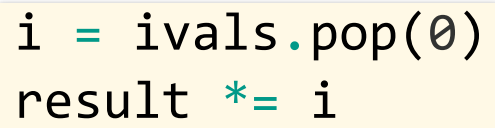
# Looping

```
(defn factorial [n]
  "computes n * (n-1) * ... * 1"
  (loop [result 1
        ival (range 2 (+ n 1))]
    (if (seq ival)
      (recur (* result (first ival))
             (rest ival))
      result)))
```



Passed by value  
to next iteration

```
def factorial(n):
    '''computes n * (n - 1) * ... * 1'''
    result = 1
    ival = range(2, n + 1)
    while ival:
        i = ival.pop(0)
        result *= i
    return result
```



Mutated in place



# Looping

```
(defn factorial [n]
  "computes n * (n-1) * ... * 1"
  (loop [result 1
        ivals (range 2 (+ n 1))]
    (if (seq ivals)
      (recur (* result (first ivals))
             (rest ivals))
      result)))
```

```
(factorial 10000)
; => 40238726007709377354370243392300398571937486421071463
;    25437999104299385123986290205920442084869694048004799
;    88610197196058631666872994808558901323829669944590997
;    ...
```

# Looping

```
(defn factorial [n]
  "computes n * (n-1) * ... * 1"
  (loop [result 1
        ival (range 2 (+ n 1))]
    (if (seq ival)
        (recur (* result (first ival))
               (rest ival))
        result)))
```

Can split into  
separate function

```
(factorial 10000)
; => 40238726007709377354370243392300398571937486421071463
;    25437999104299385123986290205920442084869694048004799
;    88610197196058631666872994808558901323829669944590997
;    ...
```

# Looping

```
(defn floop
  "inner loop for factorial"
  [result ival]
  (if (seq ival)
    (floop (* result (first ival))
            (rest ival))
    result))
```

```
(defn factorial [n]
  "computes n * (n-1) * ... * 1"
  (floop 1
        (range 2 (+ n 1))))
```

# Looping

```
(defn floop
  "inner loop for factorial"
  [result ival]
  (if (seq ival)
    (floop (* result (first ival))
            (rest ival))
    result))
```

```
(defn factorial [n]
  "computes n * (n-1) * ... * 1"
  (floop 1
        (range 2 (+ n 1))))
```

```
(factorial 10000)
; => StackOverflowError
;    clojure.lang.Numbers.equal (Numbers.java:216)
```

# Looping

```
(defn floop
  "inner loop for factorial"
  [result ival]
  (if (seq ival)
    (floop (* result (first ival))
           (rest ival))
    result))
```

Tail call

```
(defn factorial [n]
  "computes n * (n-1) * ... * 1"
  (floop 1
        (range 2 (+ n 1))))
```

# Looping

```
(defn loop
  "inner loop for factorial"
  [result ival]
  (if (seq ival)
      (recur (* result (first ival))
             (rest ival))
      result))
```

**recur** allows tail  
call optimization

```
(defn factorial [n]
  "computes n * (n-1) * ... * 1"
  (loop 1
        (range 2 (+ n 1))))
```

# Looping

```
(defn loop
  "inner loop for factorial"
  [result ivals]
  (if (seq ivals)
      (recur (* result (first ivals))
             (rest ivals))
      result))
```

**recur** allows tail  
call optimization

```
(defn factorial [n]
  "computes n * (n-1) * ... * 1"
  (loop 1
        (range 2 (+ n 1))))
```

```
(factorial 10000)
; => 40238726007709377354370243392300398571937486421071463
;    25437999104299385123986290205920442084869694048004799
;    88610197196058631666872994808558901323829669944590997
;    ...
```

# Up Next: Exercises

```
$ lein gorilla
```

