Introduction to Functional Programming and Clojure

Jan-Willem van de Meent

Anatomy of a Clojure Program

```clojure
(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

```clojure
(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)))))
```

Namespace
declaration

Anatomy of a Clojure Program

```clojure
(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)))))
```

Recursive function

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: build uberjar and run via java
lein uberjar
java -cp target/uberjar/examples-0.1.0-SNAPSHOT.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: build uberjar and run via java
lein uberjar
java -cp target/uberjar/examples-0.1.0-SNAPSHOT.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: build uberjar and run via java
lein uberjar
java -cp target/uberjar/examples-0.1.0-SNAPSHOT.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: build uberjar and run via java
lein uberjar
java -cp target/uberjar/examples-0.1.0-SNAPSHOT.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

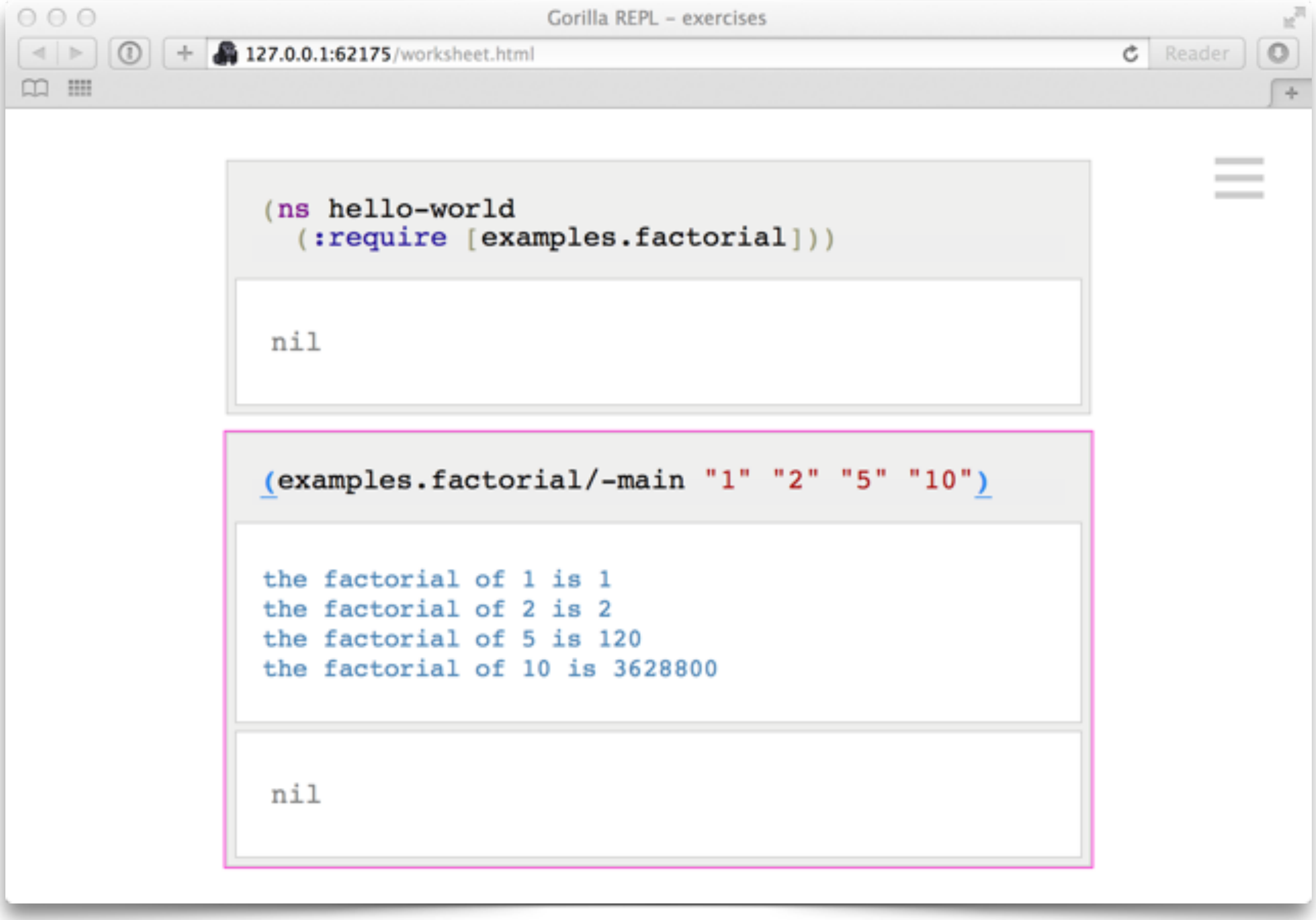
```
$ lein repl
# => nREPL server started on port 50240 on host
     127.0.0.1 - nrepl://127.0.0.1:50240
# => REPL-y 0.3.7, nREPL 0.2.12
# => Clojure 1.8.0
# => Java HotSpot(TM) 64-Bit Server VM 1.8.0-b132
# =>      Docs: (doc function-name-here)
# =>            (find-doc "part-of-name-here")
# =>    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
```



```
(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

```clojure
(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

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

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

Anatomy of a Clojure Function

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

Name

```python
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

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

Arguments

```python
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)
```

Anatomy of a Clojure Function

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

S-expression

```python
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)))))
```

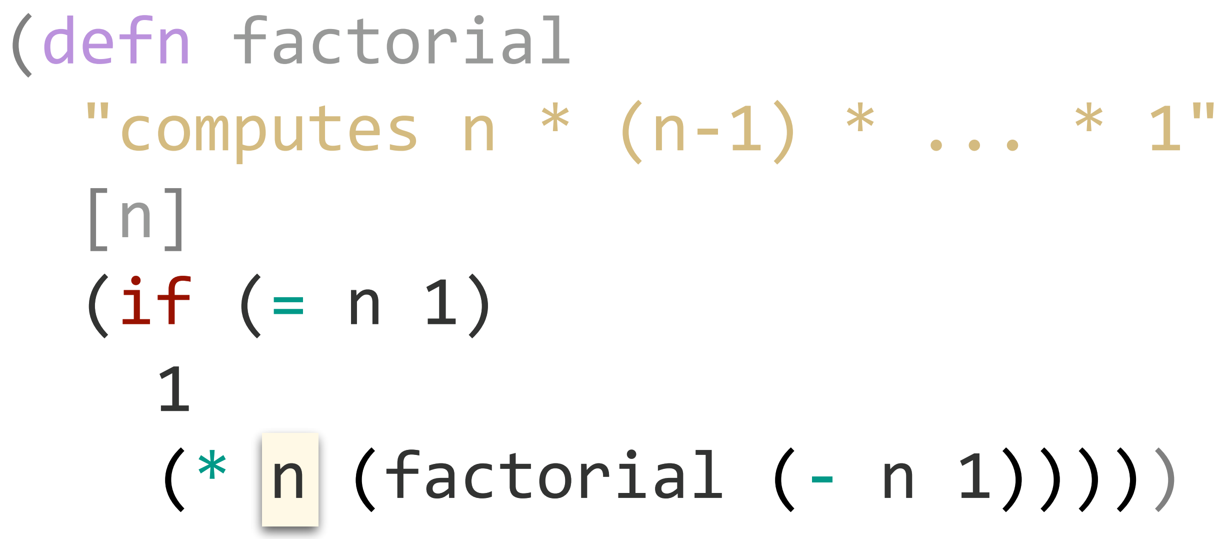
Anatomy of an Expression

```
(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)))))
```

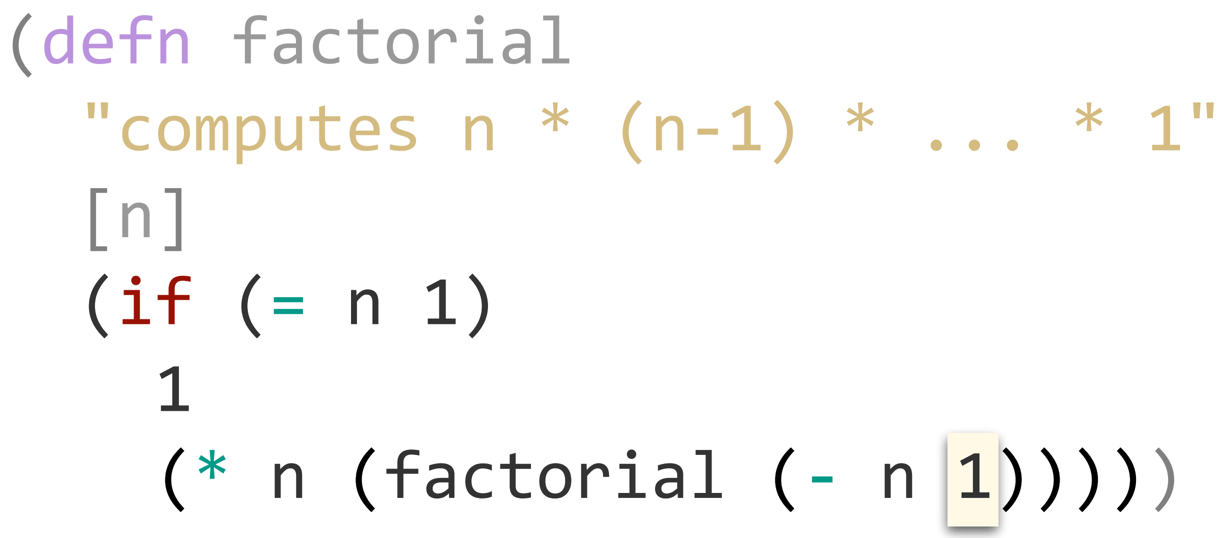
Anatomy of an Expression

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

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

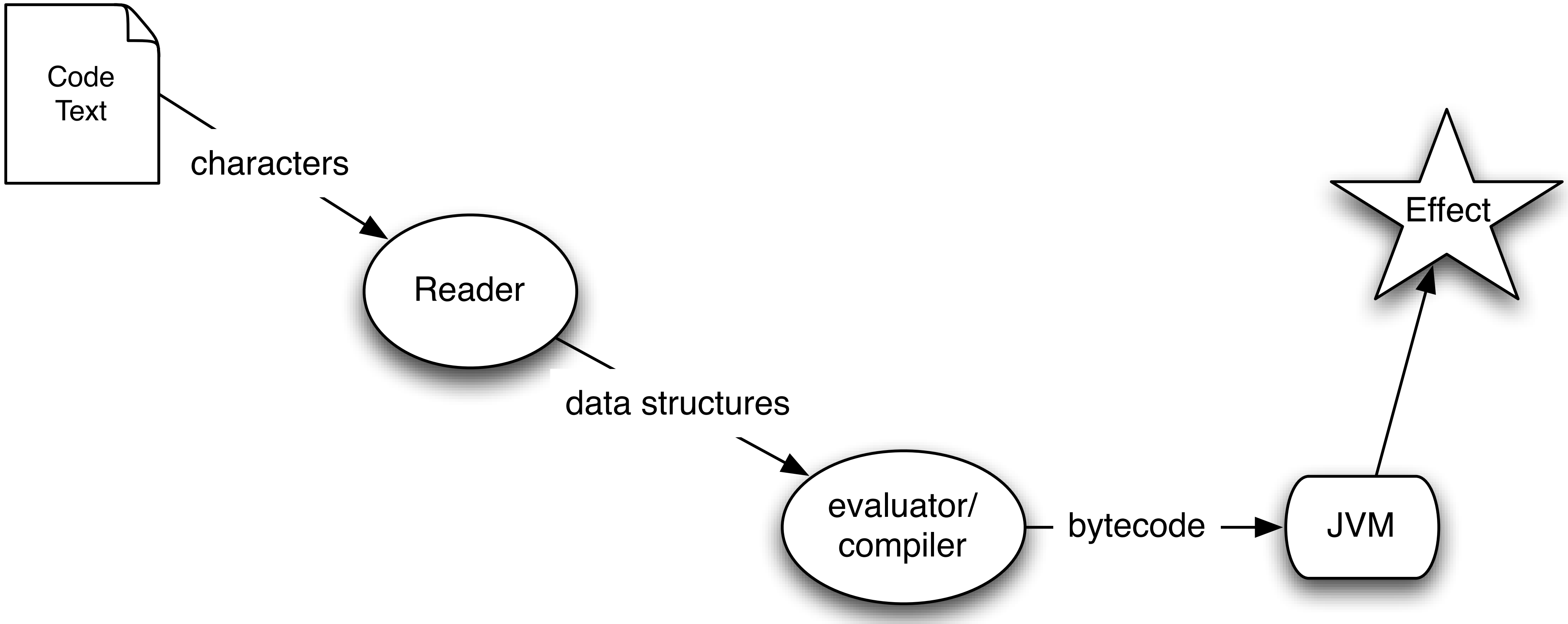
Atomic

```
;; 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}}
```

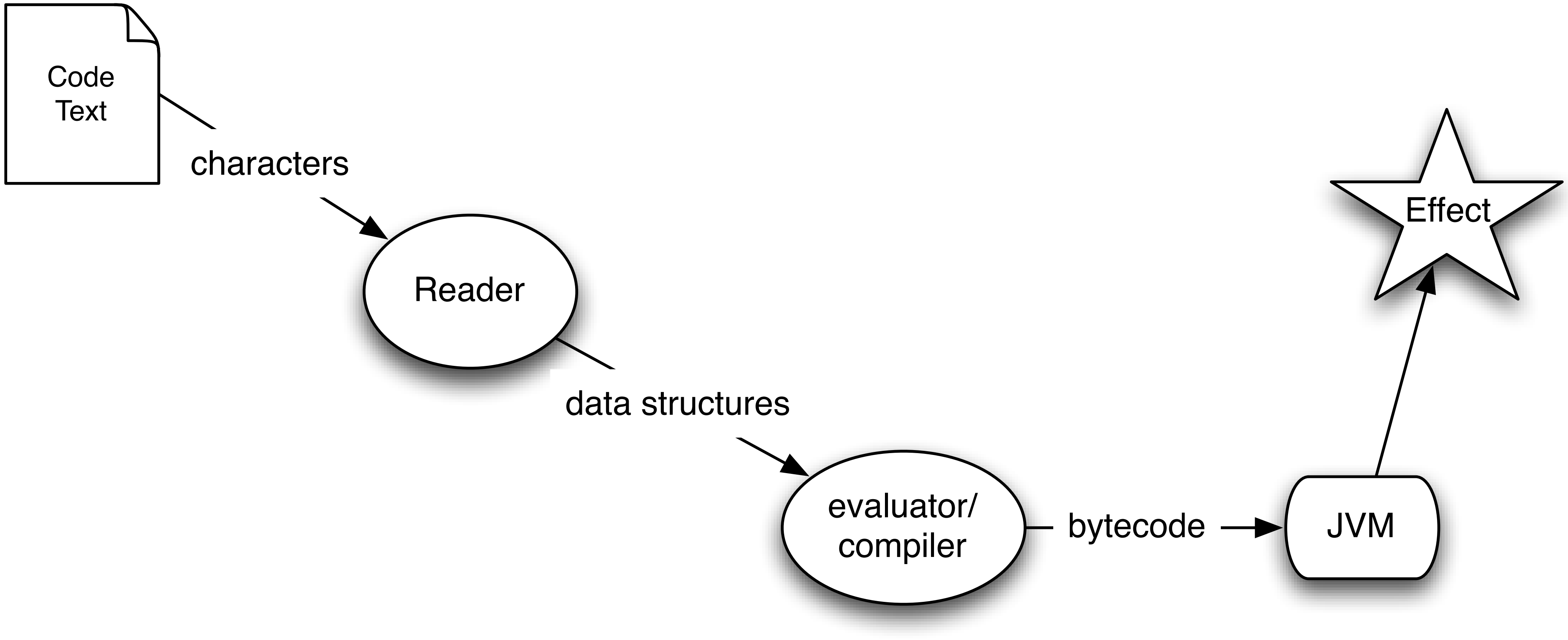
Evaluation in Clojure



```
(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)

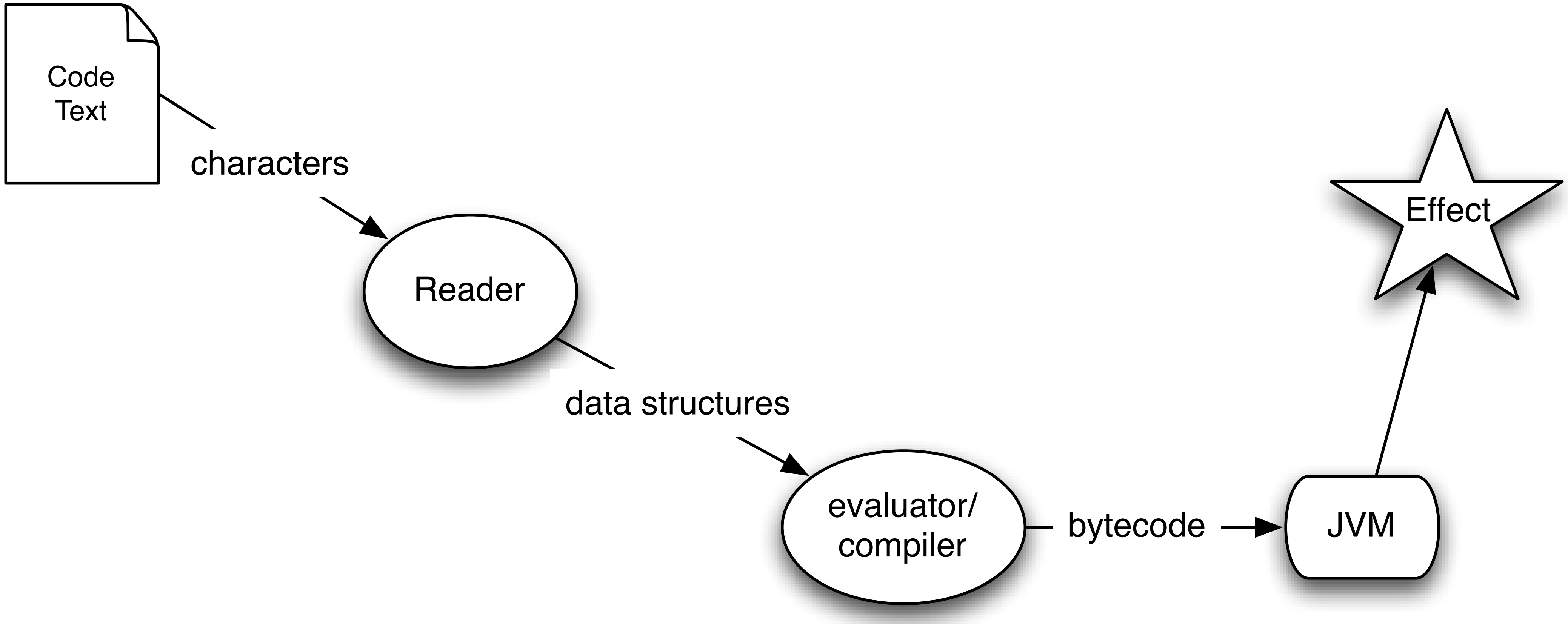
Evaluation in Clojure



```
(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)

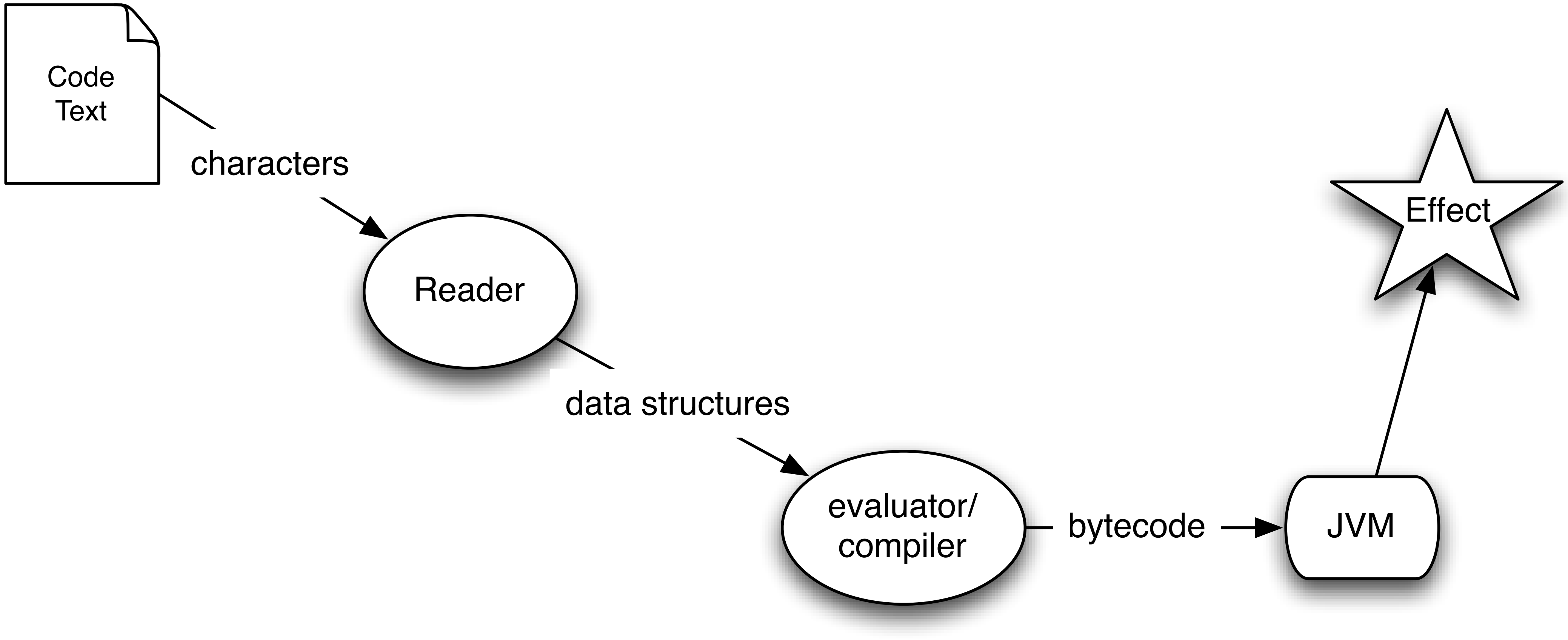
Evaluation in Clojure



```
(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)

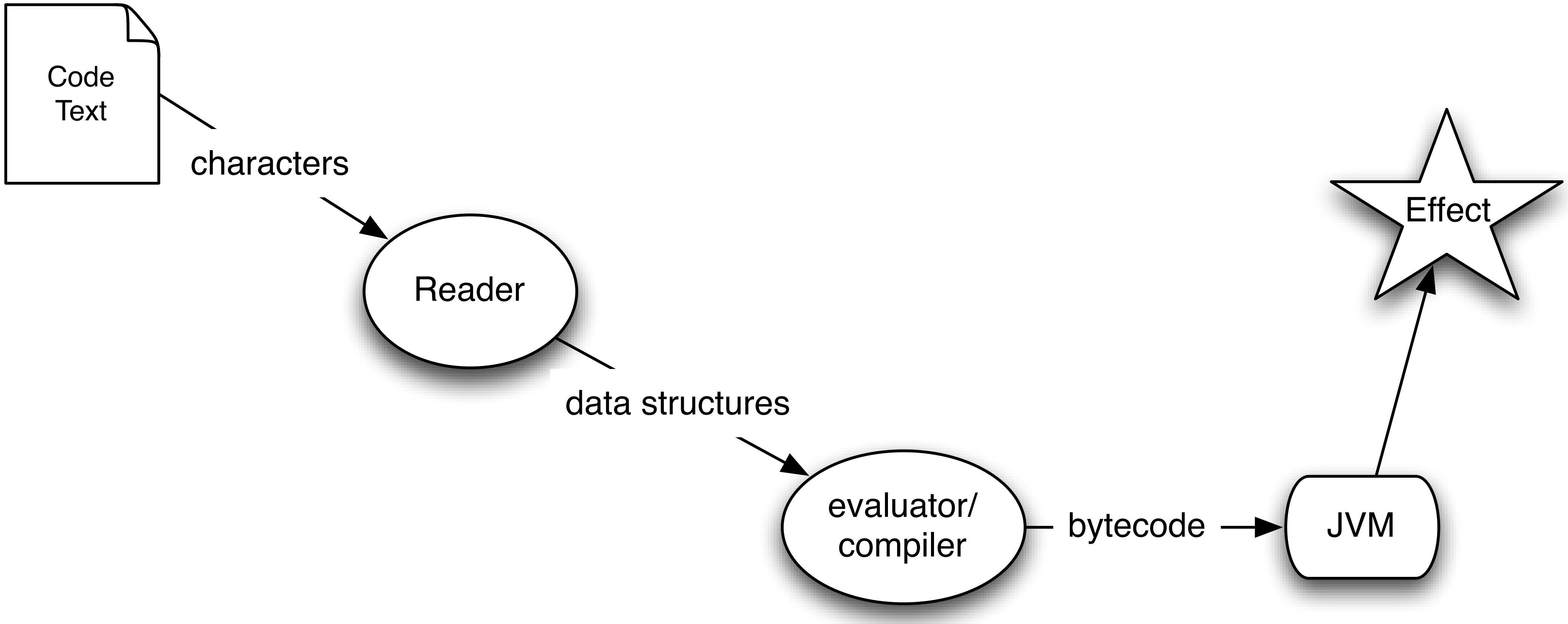
Evaluation in Clojure



```
(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)

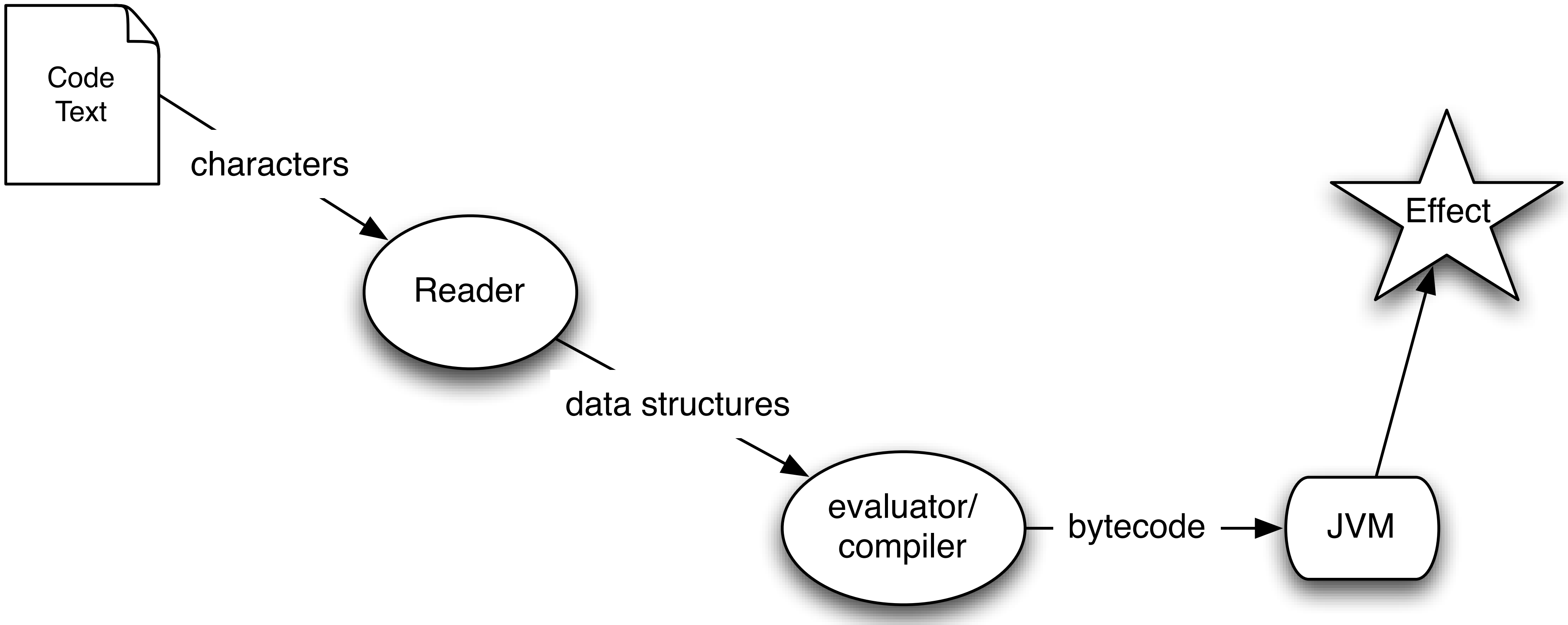
Evaluation in Clojure



```
(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)

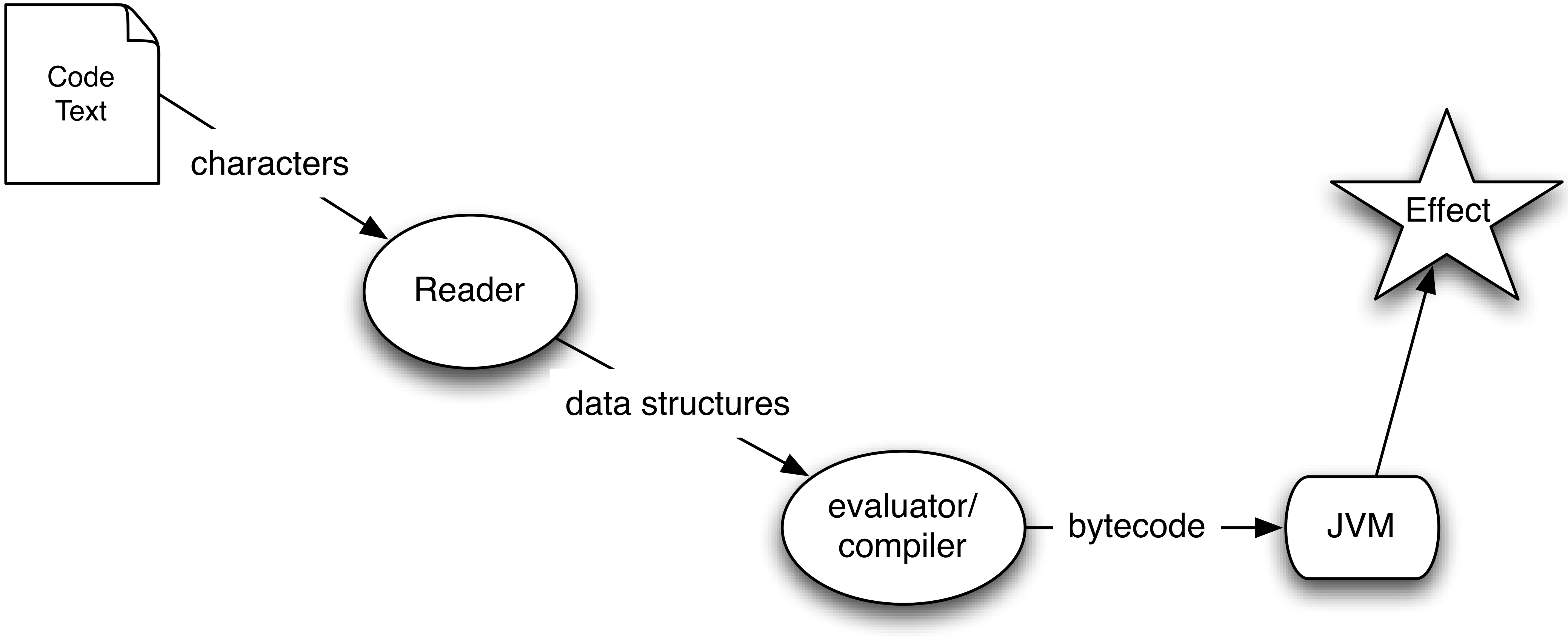
Evaluation in Clojure



```
(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)

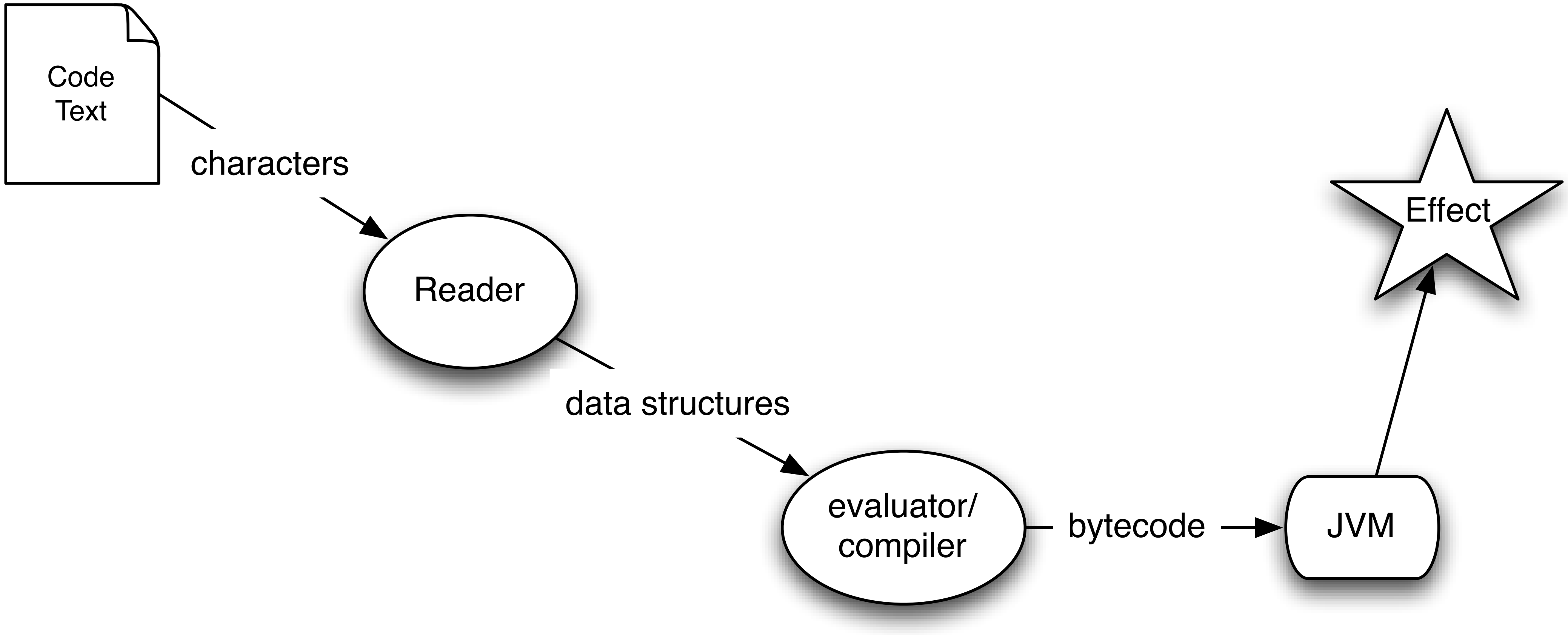
Evaluation in Clojure



```
(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)

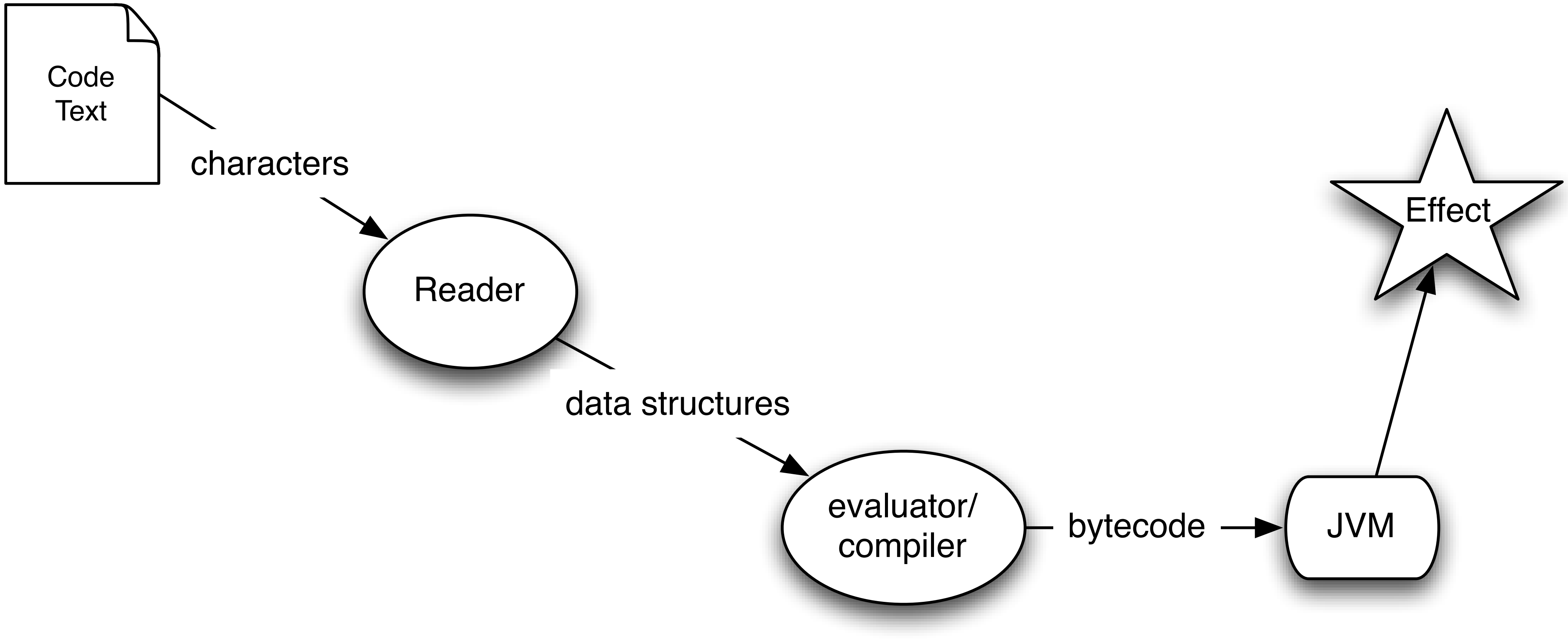
Evaluation in Clojure



```
(let [expr '(+ 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 (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)

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
```

Looping

```clojure
(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

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

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

Looping

```
(defn

  [
  (

    (
```

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

Looping

```
(defn

  [
  (

    (
```

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

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)))
```

```
def

    result
    ivals

        i
        result
```

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)))
```

Start loop

```
def

    result
    ivals

        i
        result
```

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)))
```

Initial values

```
def


    result
    ivals


        i
        result
```

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)))
```

Any values
for **i** remaining?

```
def

    result
    ivals

        i
        result
```

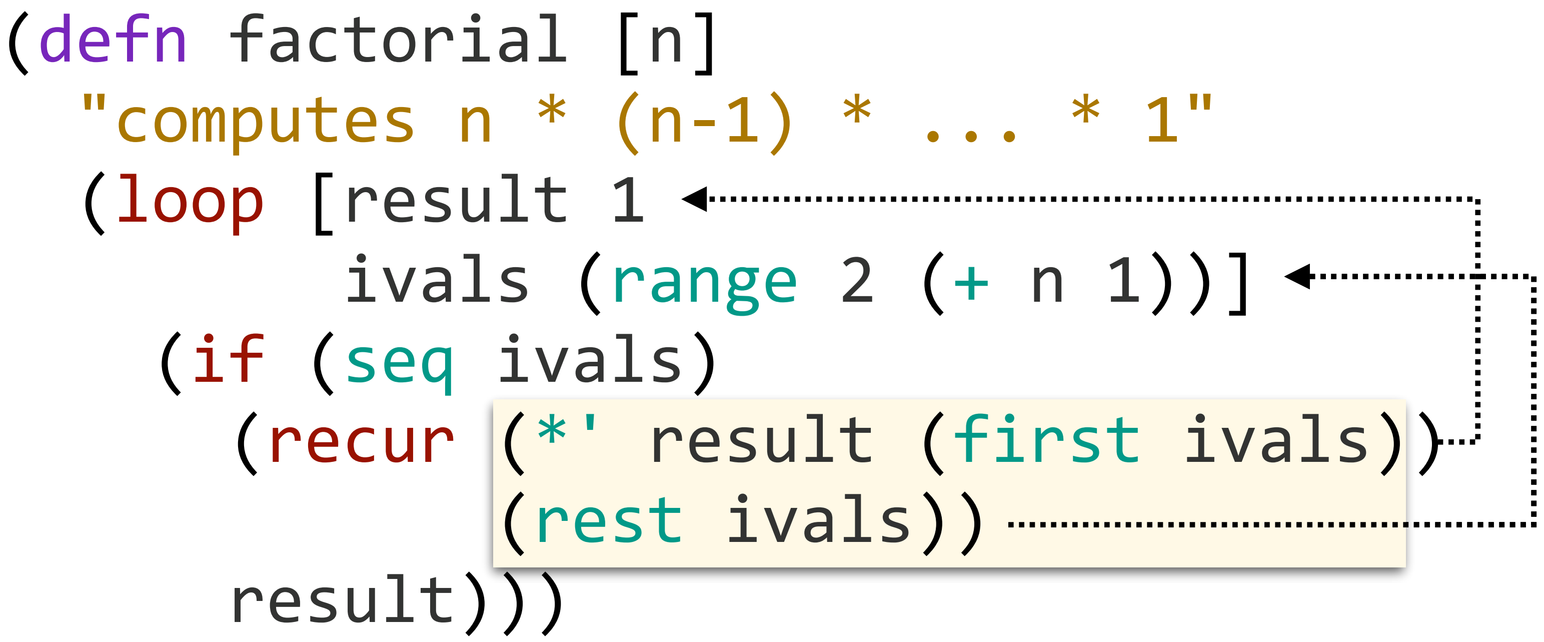
Looping

```
(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)))
```

Compute values
for next iteration

```
def


    result
    ivals

        i
        result
```

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)))
```

Compute values for next iteration

```
def

    result
    ivals

        i
        result
```

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)))
```

Output

```
def

    result
    ivals

        i
        result
```

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)))
```

```
def

    result
    ivals

        i
        result
```

Looping

```clojure
(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)))
```

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

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)))
```

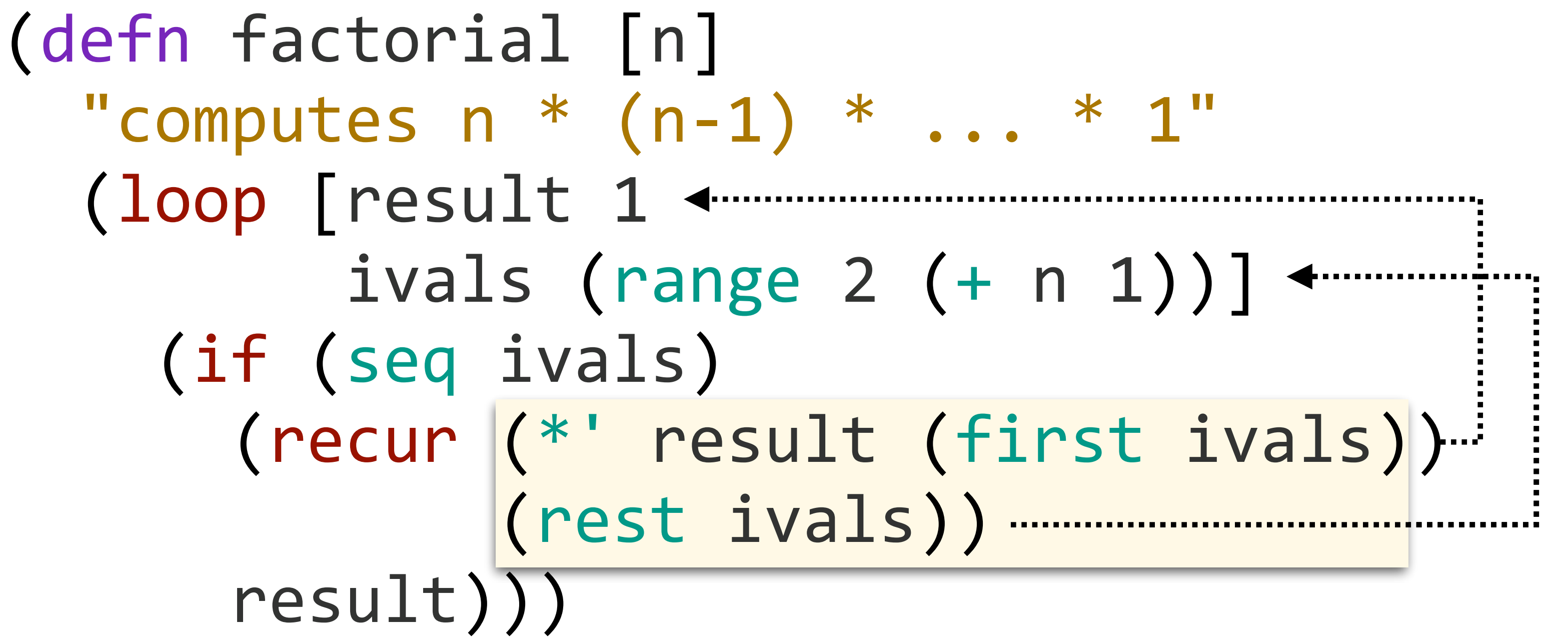
```
def factorial(n):
    '''computes n * (n - 1) * ... * 1'''
    result = 1
    ivals = range(2, n + 1)
    while ivals:
        i = ivals.pop(0)
        result *= i
    return result
```

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)))
```

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

Looping



```clojure
(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)))
```

Passed by value to next iteration

```python
def factorial(n):
    '''computes n * (n - 1) * ... * 1'''
    result = 1
    ivals = range(2, n + 1)
    while ivals:
        i = ivals.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
         ivals (range 2 (+ n 1))]
    (if (seq ivals)
      (recur (*' result (first ivals))
             (rest ivals))
      result)))
```

Can split into separate function

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

Looping

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

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

Looping

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

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

Tail call

Looping

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

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

`recur` allows tail call optimization

Looping

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

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

(factorial 10000)
; => 402387260077093773543702433923003985719374864210714632
;    2543799910429938512398629020592044208486969404800479
;    8861019719605863166687299480855890132382966994459099
;    ...
```

recur allows tail call optimization

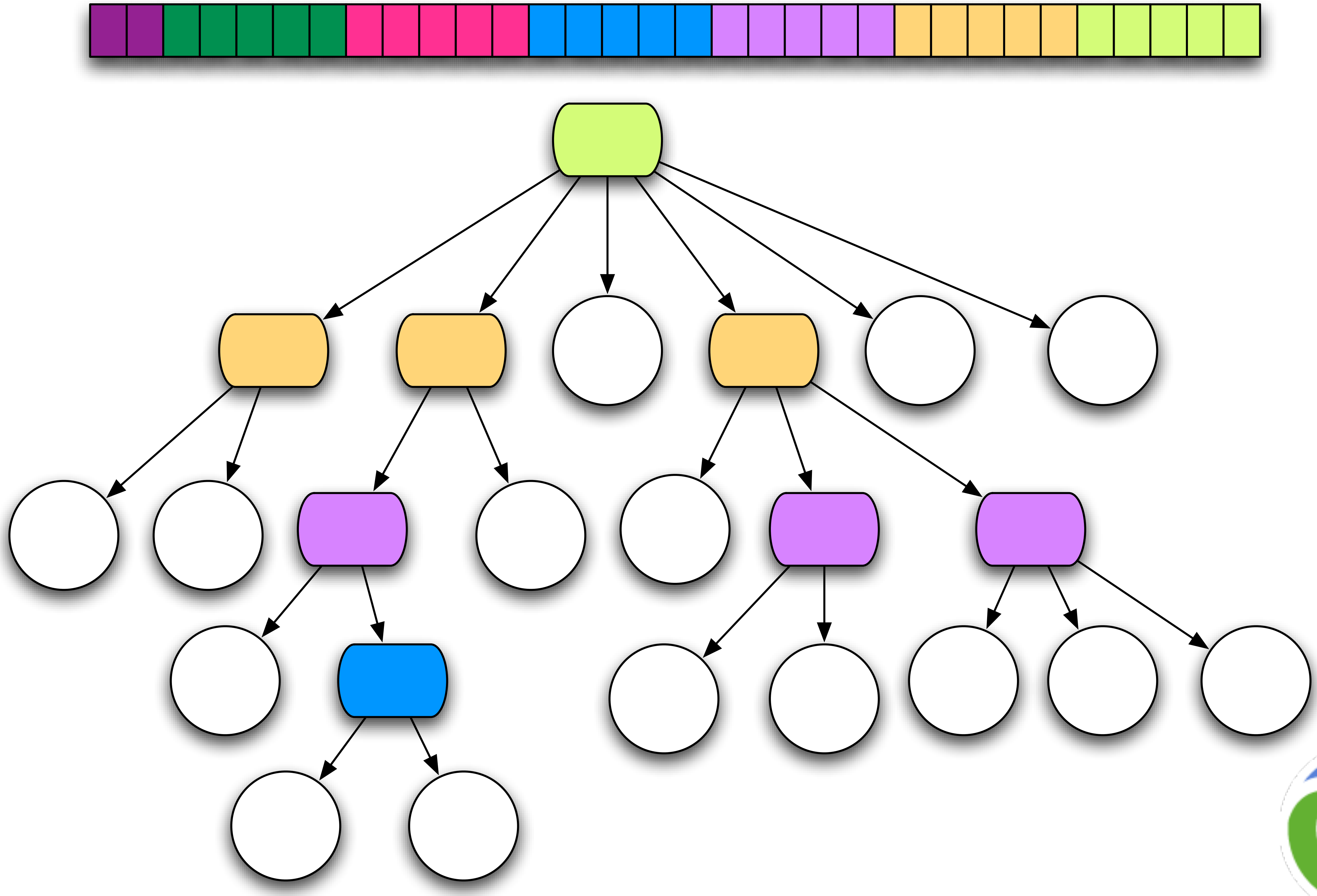
Looping

```
(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)))
```

Looping

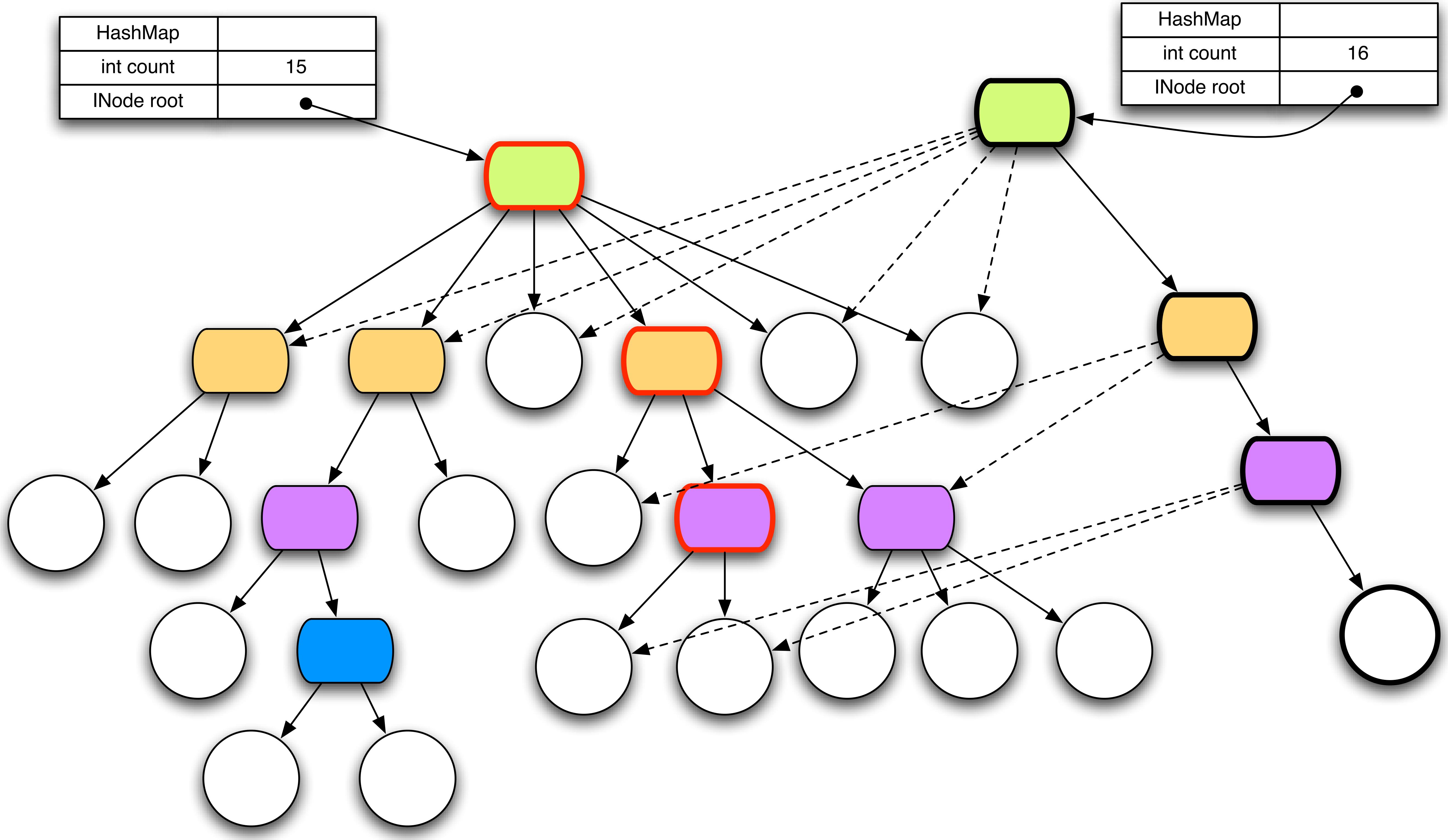
```
(defn factorial [n]
  "computes n * (n-1) * ... * 1"
  (loop [result 1
         ivals (range 2 (+ n 1))]
    (if (seq ivals)
      (recur (*' result (first ivals))
             (rest ivals))
      result)))
```

Bit-partitioned Hash Tries



(image credit: Rich Hickey)

Path Copying



(image credit: Rich Hickey)

Macros

```
(defmacro dbg
  "Prints an expression and
  its value for debugging."
  [expr]
  (list 'do
    (list 'println
      "[dbg]"
      (list 'quote expr)
      expr)
    expr))
```

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

(macroexpand '(dbg (+ 1 2))
; => (do
;       (println "[dbg]"
;                (quote (+ 1 2))
;                (+ 1 2))
;       (+ 1 2))
```

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__)
```