Introduction to Functional Programming and Clojure

Jan-Willem van de Meent
An 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))

(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)))))

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

(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))))

(ns examples.factorial (:gen-class))
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
# 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
$ 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

eamples.core=>
Interactive Shell: the REPL

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

```
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))))
```

```
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))))

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))))
)
```

```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)))))

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

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

(104x554)
(118x554)defn
(189x456)factorial
(104x488)[n]
(147x488)if
(189x456)(= n 1)
(161x456)1
(104x391)(= n 1)
(175x391)1
(104x391)(* n factorial (- n 1)))

Anatomy of an Expression

```clojure
(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)))))

(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

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

\[
(\text{defn factorial}
    \quad "\text{computes } n \times (n-1) \times \ldots \times 1"
    \quad [n]
    \quad (\text{if } (= n 1)
      \quad 1
      \quad (* n (\text{factorial} (- n 1)))))
\]

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

Anatomy of an Expression

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

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

(defn factorial
  "computes \( n \times (n-1) \times \ldots \times 1 \"
  [n]
  (if (= n 1)
    1
    (* n (factorial (- n 1)))))

expression ::= symbol | literal | (operator ...)
(defn factorial
  "computes n * (n-1) * ... * 1"
  [n]
  (if (= n 1)
    1
    (* n (factorial (- n 1)))))

expression ::= symbol | literal | (operator ...)
Anatomy of an Expression

(\texttt{defn factorial}
 \ "computes n * (n-1) * ... * 1"
 \ [n]
 \ (\texttt{if} (= n 1)
 \ 1
 \ (* n (factorial (- n 1)))))

expression ::= symbol \mid \text{literal} \mid (\text{operator} \ldots)

operator ::= special \mid \text{function} \mid \text{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

(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

(defun factorial
  "computes \text{n} \times (\text{n}-1) \times \ldots \times 1"
  \text{n}
  (if (= \text{n} 1)
    1
    (* \text{n} (factorial (- \text{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
(s 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)
(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

```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)
(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)
(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
(defn factorial
  "computes n * (n-1) * ... * 1"
[n]
(if (= n 1)
  1
  (* n (factorial (- n 1)))))
(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)
(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 21)
; => ArithmeticException integer overflow
;    clojure.lang.Numbers.throwIntOverflow (Numbers.java:1501)
Looping

(defun 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)
(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)
def factorial(n):
    '''computes n * (n - 1) * ... * 1'''
    result = 1
    for i in range(2, n + 1):
        result *= i
    return result
(defn factorial
  
  [n]
  
  (defn factorial
    
    ''computes n * (n - 1) * ... * 1''
    result = 1
    ivals = range(2, n + 1)
    while ivals:
      i = ivals.pop(0)
      result *= i
    return result
(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))))
(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)))
(defn factorial [n]
  "computes \(n \times (n-1) \times \cdots \times 1\"
  (loop [result 1
         ivals (range 2 (+ n 1))]
    (if (seq ivals)
        (recur (*' result (first ivals))
               (rest ivals))
        result))))

result
ivals
i
result
(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))))
(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
(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)))
(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
(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)))

result
ivals

i
result
(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
  ival = range(2, n + 1)
  while ival:
    i = ival.pop(0)
    result *= i
  return result
(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

```
(\texttt{defn factorial [n]} \\
\texttt{"computes n \times (n-1) \times \ldots \times 1"} \\
(\texttt{loop [result 1} \\
  \texttt{ivals (range 2 (+ n 1))]} \\
(\texttt{if (seq ival}\texttt{s}} \\
 (\texttt{recur (* result (first ival}\texttt{s}))} \\
 (\texttt{rest ival}\texttt{s})} \\
\texttt{result})))
```

```
def factorial(n):
  '''computes n \times (n - 1) \times \ldots \times 1'''
  result = 1
  ival\texttt{s} = range(2, n + 1)
  while ival\texttt{s}:
    i = ival\texttt{s}.pop(0)
    result *= i
  return result
```
(defn factorial [n]
  "computes n * (n-1) * ... * 1"
(loop [result 1]
  (vals (range 2 (+ n 1)))]
(if (seq vals)
  (recur (*' result (first vals))
    (rest vals))
  result)))

def factorial(n):
  '''computes n * (n - 1) * ... * 1'''
  result = 1
  vals = range(2, n + 1)
  while vals:
    i = vals.pop(0)
    result *= i
  return result
(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  
; 254379999104299385123986290205920442084869694048004799  
; 88610197196058631666872994808558901323829669944590997  
; ...
(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 loop
  "inner loop for factorial"
  [result ivals]
  (if (seq ivals)
    (loop (*' result (first ivals))
      (rest ivals))
    result))

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

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

(factorial 10000)
; => StackOverflowError
;   clojure.lang.Numbers.equal (Numbers.java:216)
(defn loop  
  "inner loop for factorial"  
  [result ivals]  
  (if (seq ivals)  
    (loop (*' result (first ivals))  
          (rest ivals))  
    result))

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

```clojure
(defn loop
  "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)
; => 40238726007709377354370243392300398571937486421071463
; 25437999104299385123986290205920442084869694048004799
; 88610197196058631666872994808558901323829669944590997
; ...
```

`recur` allows tail call optimization
(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)))
(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)
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))
(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__)