CS 596 Functional Programming and Design
Fall Semester, 2014
Doc 21 Macros & Monads
Nov 20, 2014

AppsFlyer

Mobile Analytics Company

Based in San Francisco

2 Billion events per day

Traffic double in 3 months

Grew from 6 to 50 people past year

Technologies used

- Redis, Kafka, Couchbase, CouchDB, Neo4j
- ElasticSearch, RabbitMQ, Consul, Docker, Mesos
- MongpDB, Riemann, Hadoop, Secor, Cascalog, AWS

AppsFlyer - Python Based

Started code base in Python

After two years python could not handle the traffic

Problems caused by

- String manipulations
- Python memory management

Their options

Rewrite parts in C & wrap in Python

Rewrite in programming language more suitable for data proccessing

- Wanted to try Functional Programming

Scala vs. OCaml vs. Haskell vs. Clojure

Scala
- Functional & Object Oriented
- They wanted pure Functional

OCaml
- Smaller community
- Only one thread runs at a time even on multicore

Haskell
- Monads made us cringe in fear

Clojure
- Runs on JVM
- Access to mutable state if needed
- Now have 10 Clojure engineers

Monads

What are they?

Why do they make engineers cringe in fear?

Function Basics

```
(println (+ 1 2)  (+ 4 5) )
```

What does this print out and why?

Function Basics

```
(and (println "A") (println "B"))
```

What does this print out and why?

Function Basics

```
(def x 5)
(def y 10)
(if (< x y)  (+ x y)  (sdsu-palindrome y))
```

Why does the if statement return a value?

Function Basics

```
(-> 25 (+ 3) Math/sqrt)
```

Control Structures - Lisp, Smalltalk

Meta

Metadata

Data about data

- Type declarations
    - public void foo()
- Java annotations

Adding Metadata

```
(def a [1 2 3])

(def b (with-meta [1 2 3] {:foo true}))

(def c ^{:foo true} [1 2 3])

(def d ^:foo [1 2 3])
```

Clojure metadata is a map

If map has one value & boolean
- Shorten to ^:key

```
(= a b c d)             true

(identical? a b)        false

(identical? b c)        false

(meta b)                {:foo true}

(meta c)                {:end-column 28, :column 21, :line 121, :foo true, :end-line 121}

(meta a)                {:end-column 15, :column 8, :line 119, :end-line 119}
```

Private, Dynamic is Metadata

```
(defn- foo [] “Example”)

(defn ^:private foo [] “Example”)

(defn ^{:private true} foo [] “Example)
```

So are Doc comments

```
(defn foo
  "A comment"
  [] 5)

(meta #'foo)

            {:ns #<Namespace basiclectures.webcrawler.basic>, :name
            foo, :file "/Users/whitney/Courses/596/Fall14/CodeExamples/
            basiclectures/src/webcraweler/basic.clj", :end-column 10, :column
            1, :line 130, :end-line 130, :arglists ([]), :doc "A comment"}
```

Macros

Clojure Data Structures & Evaluation

- Literals
  - Evaluate to themselves
  - 1 “cat” 23.4

- Symbols
  - Resolve to a value in a var
  - (def foo 5)

- Lists
  - (defn bar [x] (inc x))
  - Calls to
    - Function
    - Special form
    - Macro

Special Forms

Evaluated differently

- arguments passed unevaluated

Primitive operations

def
if
do
let
letfn
quote
var
fn
loop
recur
throw
try
monitor-enter
monitor-exit

defn
defmacro
loop
for
doseq
if-let
when-let
if-some
when-some

C Macros

Textually replacement

#define INCREMENT(x) x++

y = INCREMENT(z) → y = z++

Clojure Macros

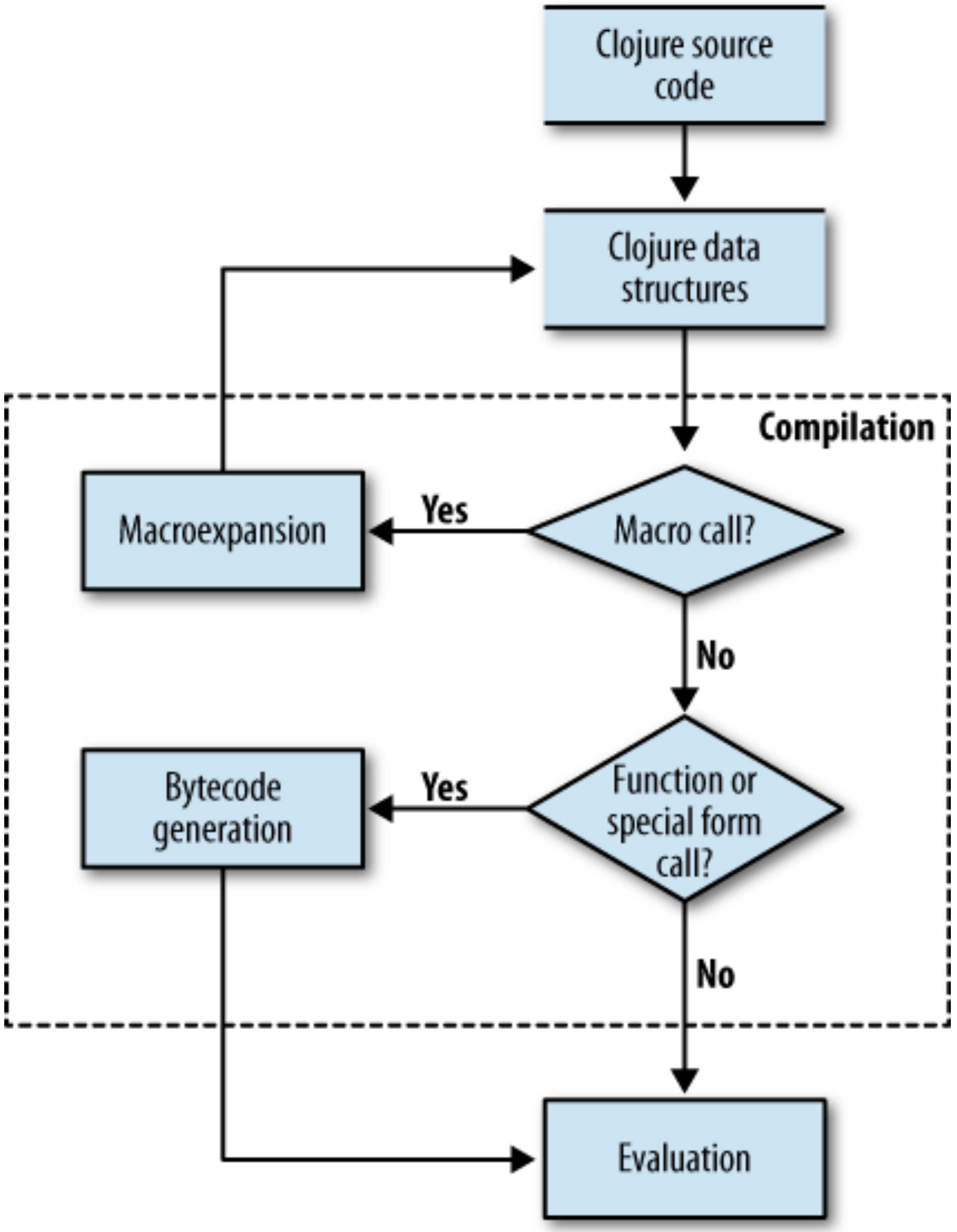
Can create their own semantics

At compile time

- Macros are given their arguments unevaluated
- Macro returns a data structure (function)

At runtime

- Macros do not exists
- Data structure returned by macro are evaluated


Clojure source code
Clojure data structures
Compilation
Macroexpansion
Yes
Macro call?
No
Bytecode generation
Yes
Function or special form call?
No
Evaluation

Note

Macros are evaluated at compile time

So runtime overhead

Macros & Special forms are not functions

```
(defn tester
   [fun]
   (fun 1 2))
```

(tester +)	3	
(tester or)	Exception	Macro
(tester if)	Exception	Special form
(tester 'or)	2	
(tester 'if)	2	

Java Motivation

```
for (int k = 0; k < foo.size(); k++) {
     x = foo.get(k);
     ...
}
```

boiler plate

Java programmers had to
live with boiler plate for 8 years

```
for (element : foo) {
     ...
}
```

Clojure macros allow you to create
own control structures

Viewing what a Macro does

macroexpand-1

Expands the macro once

macroexpand

Expands repeatedly until top level is not a macro

clojure.walk/macroexpand-all

Exapnds until there are no more macros

```
(macroexpand-1 '(cond
                   (> x y) (x - y)
                   (< x y) (y -x)))
```

```
(if (> x y)
    (x - y)
    (clojure.core/cond
        (< x y) (y -x)))
```

```
(clojure.walk/macroexpand-all '(cond
            (> x y) (x - y)
            (< x y) (y -x)))
```

```
(if (> x y)
    (x - y)
    (if (< x y)
        (y -x)
        nil))
```

```
(clojure.walk/macroexpand-all '(cond
                                 (> x y) (x - y)
                                 (< x y) (y -x)
                                 :default 0))
```

```
(if (> x y)
    (x - y)
    (if (< x y)
        (y -x)
            (if :default
                0
                nil)))
```

```
(macroexpand '(when 1 2))
```

```
(if 1 (do 2))
```

```
(macroexpand '(if 1 2))
```

```
(if 1 2)
```

```
(macroexpand '(or 1 2))
```

```
(let* [or__3975__auto__ 1]
  (if or__3975__auto__
      or__3975__auto__
   (clojure.core/or 2)))
```

When to use Macros

Remove Boilerplate code

Domain Specific Languages

Example - Testing

```
(deftest foo-test
  (is (= (foo 0) "No"))
  (is (= (foo 1) "Yes"))
  (is (= (foo 10) "Yes"))
  (is (= (foo -3) "Maybe")))
```

```
(deftest foo-test
  [input answer] (= (foo input) answer)
  0  “No”
  1  “Yes”
  10 “Yes”
  -3  “Maybe”)
```

```
(macroexpand '(are [a b c] (= a (+ b c))
                    3 2 1
                    6 1 5))


 (do
     (clojure.test/is (= 3 (+ 2 1)))
     (clojure.test/is (= 6 (+ 1 5))))
```

```
(macroexpand '(is (= 0 1)))


(try
    (clojure.core/let [values__7128__auto__ (clojure.core/list 0 1)
                       result__7129__auto__ (clojure.core/apply = values__7128__auto__)]
      (if result__7129__auto__
        (clojure.test/do-report {:type :pass, :expected (quote (= 0 1)),
               :actual (clojure.core/cons = values__7128__auto__), :message nil})
        (clojure.test/do-report {:type :fail, :expected (quote (= 0 1)),
               :actual
                  (clojure.core/list (quote not)
                     (clojure.core/cons (quote =) values__7128__auto__)), :message nil}))
                        result__7129__auto__)
    (catch java.lang.Throwable t__7156__auto__
         (clojure.test/do-report {:type :error, :expected (quote (= 0 1)),
               :actual t__7156__auto__, :message nil})))
```

Defining a Macro when

```
(defmacro when
  "Evaluates test. If logical true, evaluates body in an implicit do."
  {:added "1.0"}
  [test & body]
  (list 'if test (cons 'do body)))
```

when

```
(defmacro when
  [test & body]
  (list 'if test (cons 'do body)))
```

```
(when (= 2 (+ 1 1))
  (print "Hello")
  (println " World!"))
```

```
(list 'if
      '(= 2 (+ 1 1))
      (cons 'do
            '((print "Hello")
             (println " World!"))))
```

```
(if
      (= 2 (+ 1 1))
      (do
          ((print "Hello")
          (println " World!"))))
```

Macros

Code that produces code

list, cons and ‘ basic tools

- Cover most cases
- But awkward & lots of boilerplate

So use some macros in writing macros

Problem with Quote

```
(def a 4)

(list 1 2 3 a 5)          (1 2 3 4 5)

'(1 2 3 a 5)              (1 2 3 a 5)
```

Syntax quote `, unquote ~

```
(def a 4)

(list 1 2 3 a 5)          (1 2 3 4 5)

'(1 2 3 a 5)              (1 2 3 a 5)

`(1 2 3 ~a 5)             (1 2 3 4 5)

 '(1 2 3 ~a 5)             (1 2 3 (clojure.core/unquote a) 5)
```

Syntax quote `, unquote ~

```
(def a 4)

(def b 2)

`(1 2 4 ~(+ a b))                (1 2 4 6)
```

Inside syntax quote
unquoted elements are evaluated

Example - assert

verify the correctness of your code

```
(assert (= 1 1))          nil
(assert (= 1 2))          java.lang.AssertionError: Assert failed: (= 1 2)


(set! *assert* false)
(assert (= 1 2))               nil
```

Aside

:pre & :post conditions handle most cases were you might use assert

(set! *assert* false)

Also turns off :pre :post conditions

Example

```
(defmacro assert [x]
  (when *assert*
    `(when-not ~x
       (throw (new AssertionError (str "Assert failed: " (pr-str '~x)))))))


(macroexpand '(assert (= 1 2)))

            (if (= 1 2)
                 nil
                 (do (throw (new java.lang.AssertionError (clojure.core/str
                                                   "Assert failed: " (clojure.core/pr-str (quote (= 1
                 2))))))))
```

Namespaces, Quote ‘, Syntax Quote `

'(a b c)	(a b c)
`(a b c))	(user/a user/b user/c)

Macro Variables

```
(defmacro make-adder [x]
  `(fn [y#] (+ ~x y#)))
```

```
(def y 100)

(def add-5 (make-adder 5))

(add-5 10)
```

Macro Variables

```
(defmacro make-adder [x]
  `(fn [y#] (+ ~x y#)))
```

```
(macroexpand '(make-adder 5))
```

```
(fn* ([y__6894__auto__]
    (clojure.core/+ 5 y__6894__auto__)))
```

More Examples

```
(defmacro comment
  "Ignores body, yields nil"
  {:added "1.0"}
  [& body])

(comment
  (println "wow")
  (println "this macro is incredible"))
;=> nil

(+ 1 2) ; this is another type of comment
(+ 1 2) #_(println "this is yet another")
```

```
(defmacro try-expr [msg form]
  `(try ~(assert-expr msg form)
        (catch Throwable t#
          (do-report {:type :error, :message ~msg,
                      :expected '~form, :actual t#}))))

(defmacro is
  ([form] `(is ~form nil))
  ([form msg] `(try-expr ~msg ~form)))
```

do-while

```
(defmacro do-while [test & body]
  `(loop []
     ~@body
     (when ~test (recur))))


(defn play-game [secret]
  (let [guess (atom nil)]
    (do-while (not= (str secret) (str @guess))
      (print "Guess the secret I'm thinking: ")
      (flush)
      (reset! guess (read-line)))
    (println "You got it!")))
```

Macro Rules of thumb

Don’t create a macro when a function will do

Write an example usage

Expand your example usage by hand

Use

- macroexpand
- macroexpand-1
- clojure.walk/macroexpand-all

Experiment in REPL

Break complecated macros into smaller functions

Mastering Clojure Macros

By Colin Jones
August 26, 2014

In Safari Books online

Monoids & Monads

Monoid

Binary Function Two parameters	Integer +
Parameters and returned value have same type	2 + 1
Identity value	2 + 0
Associatively	(2+3) + 4 = 2 + (3 + 4)

Monoid

Binary Function

Two parameters

Parameters and returned value - same type

Identity value

Associatively

Java String concat

```
“hi”.concat(“ Mom”);

“hi”.concat(“”)

“hi”.concat(“Mom”.concat(“!”))
“hi”.concat(“Mom”).concat(“!”)
```

Monoid

Binary Function

- Two parameters

Parameters and returned value - same type

Identity value

Associatively

Sets union

“hi”.concat(“ Mom”);

“hi”.concat(“”)

“hi”.concat(“Mom”.concat(“!”))
“hi”.concat(“Mom”).concat(“!”)

Monoid

Associative binary function F: X*X -> X
that has an identity

Haskell

```
class Monoid m where
    mempty :: m
    mappend :: m -> m -> m
    mconcat :: [m] -> m
    mconcat = foldr mappend mempty
```