

Metaprogramming with Macros

Eugene Burmako

École Polytechnique Fédérale de Lausanne
<http://scalamacros.org/>

10 September 2012

What are macros?

Macros in programming languages:

- ▶ C macros
- ▶ Lisp macros
- ▶ ...

What is the underlying notion?

What are macros?

Macros in programming languages:

- ▶ C macros
- ▶ Lisp macros
- ▶ ...

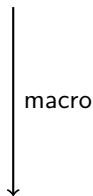
What is the underlying notion?

The notion of textual abstraction:

- ▶ Recognize pieces of text that match a specification
- ▶ Replace them according to a procedure

What are macros?

```
printf("Hello %s!", "World")
```



```
def formatter(arg1: Any) = "Hello " + arg1.toString + "!"  
print(formatter("World"))
```

Why macros?

Work with lexical tokens or syntax trees, therefore are not bound by the semantics of the underlying programming language

Use cases:

- ▶ Deeply embedded DSLs (database access, testing)
- ▶ Optimization (programmable inlining, fusion)
- ▶ Analysis (integrated proof-checker)
- ▶ Effects (effect containment and propagation)
- ▶ ...

Challenges in macrology

- ▶ Notation
- ▶ Variable capture
- ▶ Typechecking
- ▶ Syntax extensibility
- ▶ ...

The focus of this talk

Inadvertent variable capture:

- ▶ Macro expansions sometimes cause name clashes
- ▶ Some identifiers end up referring to variables from other scopes

Outline

The prelude of macros: introduces the running example

The chapter of bindings: illustrates the problem of variable capture

The trilogy of tongues: surveys macro systems that solve this problem

The vision of the days to come: presents the research proposal

A detour: how Lisp works

```
(if (calculate)
    (print "success")
    (error "does not compute"))
```

- ▶ S-expressions: atoms and lists
- ▶ print and error are one-argument functions
- ▶ calculate is a zero-argument function
- ▶ if is a special form
- ▶ All values can be used in conditions

Anaphoric if

```
(aif (calculate)
      (print it)
      (error "does not compute"))
```

```
(let* ((temp (calculate))
        (it temp))
      (if temp
          (print it)
          (error "does not compute")))
```

The aif macro

```
(aif (calculate)
      (print it)
      (error "does not compute"))
```

```
(defmacro aif args
```

```
(let* ((temp (calculate))
        (it temp))
  (if temp
      (print it)
      (error "does not compute"))))
```

Low-level implementation

```
(aif (calculate)
  (print it)
  (error "does not compute"))

(defmacro aif args
  (list 'let* (list (list 'temp (car args))
                    (list 'it 'temp))
        (list 'if 'temp
              (cadr args)
              (caddr args))))

(let* ((temp (calculate))
      (it temp))
  (if temp
    (print it)
    (error "does not compute")))
```

Quasiquoting: static template

```
(aif (calculate)
  (print it)
  (error "does not compute"))

(defmacro aif args
  '(let* ((temp .....))
    (it temp))
  (if temp
    .....
    .....))

(let* ((temp (calculate))
  (it temp))
  (if temp
    (print it)
    (error "does not compute")))
```

Quasiquoting: dynamic holes

```
(aif (calculate)
  (print it)
  (error "does not compute"))

(defmacro aif args
  `(let* ((temp ,(car args))
          (it temp))
    (if temp
      ,(cadr args)
      ,(caddr args)))

(let* ((temp (calculate))
      (it temp))
  (if temp
    (print it)
    (error "does not compute")))
```

Macro by example (MBE)

```
(aif (calculate)
  (print it)
  (error "does not compute"))

(defmacro+ aif
  (aif cond then else)
  (let* ((temp cond)
        (it temp))
    (if temp
        then
        else)))

(let* ((temp (calculate))
      (it temp))
  (if temp
    (print it)
    (error "does not compute")))
```

Interlude

```
(defmacro+ aif
  (aif cond then else)
  (let* ((temp cond)
        (it temp))
    (if temp then else)))
```

- ▶ Macros are functions that transform syntax objects
- ▶ Quasiquotes = static templates + dynamic holes

The aif macro is buggy

```
(aif (calculate)
  (print it)
  (error "does not compute"))
```

```
(defmacro+ aif
  (aif cond then else)
  (let* ((temp cond)
        (it temp))
    (if temp then else)))
```

The aif macro is buggy

```
(aif (calculate)
  (print it)
  (error "does not compute"))

(defmacro+ aif
  (aif cond then else)
  (let* ((temp cond)
        (it temp))
    (if temp then else)))

(let* ((temp (calculate))
      (it temp))
  (if temp
    (print it)
    (error "does not compute")))
```

Bug #1: Violation of hygiene

```
(let ((temp 451°F))  
  (aif (calculate)  
    (print it)  
    (print temp)))
```

```
(defmacro+ aif  
  (aif cond then else)  
  (let* ((temp cond)  
    (it temp))  
    (if temp then else)))
```

```
(let ((temp 451°F))  
  (let* ((temp (calculate))  
    (it temp))  
    (if temp  
      (print it)  
      (print temp))))
```

Bug #2: Violation of referential transparency

```
(let ((if hijacked))  
  (aif (calculate)  
    (print it)  
    (error "does not compute"))))
```

```
(defmacro+ aif  
  (aif cond then else)  
  (let* ((temp cond)  
    (it temp))  
    (if temp then else))) ;; core if
```

```
(let ((if hijacked))  
  (let* ((temp (calculate))  
    (it temp))  
    (if temp ;; hijacked if  
      (print it)  
      (error "does not compute"))))
```

Old school solution

```
(defmacro+ aif
  (aif cond then else)
  (let ((temp (gensym)))
    (let* ((temp cond)
           (it temp))
      (if temp then else))))
```

And please don't rename core forms

Three macro-enabled languages

Template Meta-programming for Haskell [\[Template Haskell\]](#)

by Tim Sheard and Simon Peyton Jones

Meta-programming in Nemerle [\[Nemerle\]](#)

by Kamil Skalski, Michal Moskal and Pawel Olszta.

Keeping it Clean with Syntax Parameters [\[Racket\]](#)

by Eli Barzilay, Ryan Culpepper and Matthew Flatt

All three languages:

- ▶ Solve the problems of hygiene and referential transparency
- ▶ Do that in their own interesting ways

Template Haskell: Introduction

```
$(aif [| calculate |]  
    [| putStrLn (show it) |]  
    [| error "does not compute" |])
```

```
aif :: Q Exp -> Q Exp -> Q Exp -> Q Exp
```

```
aif cond then' else' =
```

```
    [| let temp = $cond
```

```
        it = temp
```

```
        in if temp /= 0 then $then' else $else' |]
```

- ▶ No dedicated concept of macros
- ▶ Macro expansions are triggered explicitly with `$`
- ▶ There are quasiquotes `[| ... |]` and unquotes `$expr`
- ▶ Hygienic and referentially transparent

Template Haskell: The perils of hygiene

```
$(aif [| calculate |]  
    [| putStrLn (show it) |]  
    [| error "does not compute" |])
```

```
aif :: Q Exp -> Q Exp -> Q Exp -> Q Exp
```

```
aif cond then' else' =
```

```
    [| let temp = $cond
```

```
        it = temp
```

```
        in if temp /= 0 then $then' else $else' |]
```

```
let temp_almx = calculate
```

```
    it_almy = temp_almx
```

```
in if (temp_almx /= 0)
```

```
    then putStrLn (show it)
```

```
    else error "does not compute"
```

Not in scope: 'it'

Template Haskell: The Q monad

```
aif cond then' else' =  
  [| let temp = $cond  
      it = temp  
      in if temp /= 0 then $then' else $else' |]
```

```
aif :: Q Exp -> Q Exp -> Q Exp -> Q Exp
```

```
aif cond' then'' else'' =  
  do { ...  
      ; temp <- newName "temp"  
      ; it <- newName "it"  
      ; let notEq = mkNameG_v "ghc-prim" "GHC.Classes" "/=" in  
        return (LetE ... (Conde (... then' else')) ...)  
  }
```

Template Haskell: Breaking hygiene

```
$(aif [| calculate |]  
  [| putStrLn (show $(dyn "it")) |]  
  [| error "does not compute" |])  
  
aif :: Q Exp -> Q Exp -> Q Exp -> Q Exp  
aif cond then' else' =  
  [| let temp = $cond  
      it = temp  
      in if temp /= 0 then $then' else $else' |]  
  
let temp_almx = calculate  
  it_almy = temp_almx  
in if (temp_almx /= 0)  
  then putStrLn (show it_almy)  
  else error "does not compute"
```

Template Haskell: Summary

- ▶ In Template Haskell quasiquotes are compiled down to the Q monad
- ▶ The Q monad takes care of names
- ▶ Sometimes we need to break hygiene

Nemerle: Introduction

```
aif(calculate,  
    WriteLine(it),  
    throw Exception("does not compute"))
```

```
macro aif(cond, then, else_) {  
  <[  
    def temp = $cond;  
    def it = temp;  
    if (temp != 0) $then else $else_  
  ]>  
}
```

- ▶ Macros are declared explicitly, expansions are implicit
- ▶ There are quasiquotes <[...]> and unquotes \$expr
- ▶ Hygienic and referentially transparent

Nemerle: The perils of hygiene

```
aif(calculate,  
    WriteLine(it),  
    throw Exception("does not compute"))
```

```
macro aif(cond, then, else_) {  
  <[  
    def temp = $cond;  
    def it = temp;  
    if (temp != 0) $then else $else_  
  ]>  
}
```

```
def calculate = 42;  
def temp_1087 = calculate;  
def it_1088 = temp_1087;  
if (temp_1087 != 0) WriteLine(it) else throw Exception("...")
```

error: unbound name 'it'

Nemerle: Coloring algorithm

```
def calculate = 42;  
aif(calculate,  
    WriteLine(it),  
    throw Exception("does not compute"))
```

```
macro aif(cond, then, else_) {  
    <[  
        def temp = $cond;  
        def it = temp;  
        if (temp != 0) $then else $else_  
    ]>  
}
```

```
def calculate = 42;  
def temp = calculate;  
def it = temp;  
if (temp != 0) WriteLine(it) else throw Exception("...")
```

Nemerle: Coloring algorithm

```
def calculate = 42;                                // top-level color
aif(calculate,
    WriteLine(it),
    throw Exception("does not compute"))
```

```
macro aif(cond, then, else_) {
  <[
    def temp = $cond;
    def it = temp;
    if (temp != 0) $then else $else_
  ]>
}
```

```
def calculate = 42;
def temp = calculate;
def it = temp;
if (temp != 0) WriteLine(it) else throw Exception("...")
```

Nemerle: Coloring algorithm

```
def calculate = 42;                                // top-level color
aif(calculate,
    WriteLine(it),
    throw Exception("does not compute"))

macro aif(cond, then, else_) {                      // expansion color
    <[
        def temp = $cond;
        def it = temp;
        if (temp != 0) $then else $else_
    ]>
}

def calculate = 42;
def temp = calculate;
def it = temp;
if (temp != 0) WriteLine(it) else throw Exception("...")
```


Nemerle: Coloring algorithm

```
def calculate = 42;                                // top-level color
aif(calculate,
    WriteLine(it),
    throw Exception("does not compute"))

macro aif(cond, then, else_) {                      // expansion color
  <[
    def temp = $cond;
    def it = temp;
    if (temp != 0) $then else $else_
  ]>
}

def calculate = 42;                                // bind using colors
def temp = calculate;
def it = temp;
if (temp != 0) WriteLine(it) else throw Exception("...")
```

Nemerle: Breaking hygiene

```
def calculate = 42;                                // top-level color
aif(calculate,
    WriteLine(it),
    throw Exception("does not compute"))

macro aif(cond, then, else_) {                      // expansion color
    <[
        def temp = $cond;
        def $("it": ussite) = temp;                // recolor the variable
        if (temp != 0) $then else $else_
    ]>
}

def calculate = 42;                                // bind using colors
def temp = calculate;
def it = temp;
if (temp != 0) WriteLine(it) else throw Exception("...")
```

Nemerle: Summary

- ▶ Nemerle takes care of hygiene with a coloring algorithm
- ▶ No complex translation algorithms are necessary
- ▶ As another bonus programmer can fine-tune colors with `MacroColors`
- ▶ Referential transparency works as well

Racket: Introduction

```
(aif (calculate)
  (print it)
  (error "does not compute"))

(define-syntax (aif stx)
  (syntax-case stx ()
    ((aif cond then else)

      #'(let ((temp cond)
              (it temp)))
        (if temp then else))))
```

- ▶ A Lisp, descendent from Scheme
- ▶ 25 years of hygienic macros, a bunch of macro systems
- ▶ Language features written using macros (classes, modules, etc)

Racket: The perils of hygiene

```
(aif (calculate)
  (print it)
  (error "does not compute"))

(define-syntax (aif stx)
  (syntax-case stx ()
    ((aif cond then else)

      #'(let ((temp cond)
              (it temp)))
      (if temp then else))))

(let* ((temp (calculate))
      (it temp))
  (if temp
    (print it)
    (error "does not compute")))
```

Racket: Breaking hygiene

```
(aif (calculate)
  (print it)
  (error "does not compute"))

(define-syntax (aif stx)
  (syntax-case stx ()
    ((aif cond then else)
     (with-syntax ((it (datum->syntax #'aif 'it)))
       #'(let ((temp cond)
               (it temp)))
           (if temp then else))))))

(let* ((temp (calculate))
      (it temp))
  (if temp
    (print it)
    (error "does not compute")))
```

Racket: The `aunless` macro

```
(aunless (not (calculate))
  (print it)
  (error "does not compute"))

(define-syntax (aunless stx)
  (syntax-case stx ()
    ((aunless cond then else)
     #'(aif (not cond) then else))))

(let* ((temp (not (not (calculate))))
      (it temp))
  (if temp
    (print it)
    (error "does not compute")))
```

Racket: Being unhygienic doesn't scale

```
(unless (not (calculate))
  (print it)
  (error "does not compute"))

(define-syntax (unless stx)
  (syntax-case stx ()
    ((unless cond then else)
     #'(aif (not cond) then else))))

(let* ((temp (not (not (calculate))))
      (it temp))
  (if temp
    (print it)
    (error "does not compute")))
```


Racket: Being unhygienic doesn't scale

```
(unless (not (calculate))
  (print it)
  (error "does not compute"))

(define-syntax (unless stx)
  (syntax-case stx ()
    ((unless cond then else)
     #'(aif (not cond) then else))))

(let* ((temp (not (not (calculate))))
      (it temp))
  (if temp
    (print it)
    (error "does not compute")))
```

Racket: In a search for a better solution

What we are doing:

- ▶ We're trying to introducing a variable that transcends scopes
- ▶ And we're doing this by manually passing this variable around

Racket: In a search for a better solution

What we are doing:

- ▶ We're trying to introducing a variable that transcends scopes
- ▶ And we're doing this by manually passing this variable around

How we can do better:

- ▶ The same problem is already solved in Lisp at runtime level
- ▶ The solution is to use dynamic variables
- ▶ We can try to marry this language feature with macros

Racket: Syntax parameters

```
(define-syntax-parameter it (syntax-rules ()))
```

```
(define-syntax (aif stx)
  (syntax-case stx ()
    ((aif cond then else)
     #'(let ((temp cond))
         (syntax-parameterize
          ((it (syntax-rules () ((_ temp))))
           (if temp then else))))))
```

- ▶ it becomes a compile-time dynamic variable
- ▶ Therefore its scope overarches all potential expansions
- ▶ High-level language feature (dynamic variables) + macros = win

Summary

Macros:

- ▶ Macros provide impressive power for their simplicity
- ▶ But they also give rise to unusual problems
- ▶ One of these problems involves mixed up bindings

Summary

Macros:

- ▶ Macros provide impressive power for their simplicity
- ▶ But they also give rise to unusual problems
- ▶ One of these problems involves mixed up bindings

Bindings:

- ▶ Automatic hygiene and referential transparency are real
- ▶ Sometimes it is necessary to break hygiene
- ▶ There are ways of doing that
- ▶ Sometimes these ways are too low-level

Summary

Macros:

- ▶ Macros provide impressive power for their simplicity
- ▶ But they also give rise to unusual problems
- ▶ One of these problems involves mixed up bindings

Bindings:

- ▶ Automatic hygiene and referential transparency are real
- ▶ Sometimes it is necessary to break hygiene
- ▶ There are ways of doing that
- ▶ Sometimes these ways are too low-level

Future work:

- ▶ Integration with other language features provides unexpected insights

Scala macros

- ▶ Since this spring Scala has macros
- ▶ Even better: macros are an official part of the language in the next production release 2.10.0
- ▶ Now it's time to put the pens down and think about the future
- ▶ The future is in integration with other language features

Implicits

```
def serialize[T](x: T): Pickle
```

Implicits

```
trait Serializer[T] {  
  def write(pickle: Pickle, x: T): Unit  
}  
  
def serialize[T](x: T)(s: Serializer[T]): Pickle
```

Implicits

```
trait Serializer[T] {  
  def write(pickle: Pickle, x: T): Unit  
}  
  
def serialize[T](x: T)(implicit s: Serializer[T]): Pickle  
  
implicit object ByteSerializer extends Serializer[Byte] {  
  def write(pickle: Pickle, x: Byte) = pickle.writeByte(x)  
}
```

Implicits

```
trait Serializer[T] {  
  def write(pickle: Pickle, x: T): Unit  
}  
  
def serialize[T](x: T)(implicit s: Serializer[T]): Pickle  
  
implicit def generator: Serializer[T] = macro impl[T]  
def impl[T](c: Context): c.Expr[Serializer[T]] = ...
```

Research proposal

Marry macros and high-level language features:

- ▶ Macros + functions → programmable inlining, specialization, fusion
- ▶ Macros + annotations → code contracts, statically-typed decorators
- ▶ Macros + implicits → static verification
- ▶ ...

Backup slides

Macros for database access: SLICK

```
@table("COFFEES") case class Coffee(  
  @column("COF_NAME") name: String,  
  @column("SUP_ID") supID: Int,  
  @column("PRICE") price: Double  
)  
  
val coffees = Queryable[Coffee]  
  
val l = for { c <- coffees if c.supID == 101 }  
yield (c.name, c.price)  
  
backend.result(l, session)  
  .foreach { case (n, p) => println(n + ": " + p) }
```

- ▶ Deeply embedded domain-specific language
- ▶ Constructs like field access and method calls are overloaded
- ▶ Underlying macros save ASTs till runtime and translate them to SQL

Macros for testing: ScalaMock

```
val w = mock[Warehouse]
inSequence {
  w.expects.hasInventory("Talisker", 50).returning(true)
  w.expects.remove("Talisker", 50).once
}
```

```
val order = new Order("Talisker", 50)
order.fill(w)
assert(order.isFilled)
```

- ▶ Deeply-embedded domain-specific language
- ▶ Macro types generate mocks at compile-time
- ▶ Boilerplate generation is completely automatic

Macros for inlining: Scala collections

```
def filter(p: T => Boolean): Repr = ...
```

```
def filter(p: T => Boolean): Repr = macro inline {  
  ... the original body of filter ...  
}
```

- ▶ The `filter` function transparently becomes a macro
- ▶ This doesn't break source compatibility
- ▶ The original body of `filter` remains the same
- ▶ Yet the underlying macro is now in full control of inlining

Macros for fusion: Courtesy of Paul Phillips

```
def inc(x: Int) = x + 1
def f = List(1, 2, 3) map inc map inc map inc
def g = List(1, 2, 3) map inc map inc map inc fuse
```

- ▶ Desktop fusion achieved!
- ▶ How to deal with side effects?
- ▶ Also what about data flow analysis?

Macros for verification: Courtesy of Alexander Kuklev

```
trait SemiGroup[T] extends Eq[T] {  
  def ◦(a: T, b: T): T  
  def associativity(a: T, b: T, c: T):  
    ✓((a ◦ (b ◦ c)) == ((a ◦ b) ◦ c))  
}
```

```
def reduce[T](op: (T, T) => T): T
```

```
def reduce[T](op: (T, T) => T)(implicit evidence:  
  ✓((a: T, b: T, c: T) => op(op(a, b), c) == op(a, op(b, c)))  
): T
```

- ▶ Facts are encoded with the ✓ macro
- ▶ Proofs are requested with implicit parameters
- ▶ Proofs can either be inferred by implicit macros or provided by hand

Scala in the present: Macro defs

```
object Asserts {  
  def assertionsEnabled = ...  
  def raise(msg: Any) = throw new AssertionError(msg)  
  
  def assert(cond: Boolean, msg: Any) = macro impl  
  def impl(c: Context)  
    (cond: c.Expr[Boolean], msg: c.Expr[Any]) =  
    if (assertionsEnabled)  
      c.reify(if (!cond.eval) raise(msg.eval))  
    else  
      c.reify()  
}
```

- ▶ Separate macro definitions and implementations
- ▶ reify ensures hygiene and referential transparency
- ▶ reify also implements the notion of quasiquoting

Scala in the future: Type macros

```
type MySQLdb(connString: String) = macro ...  
  
type MyDb = Base with MySQLdb("Server=127.0.0.1")  
  
import MyDb._  
val products = new MyDb().products  
products.filter(p => p.name.startsWith("foo")).toList
```

- ▶ Generalize macros from term refs to symbol refs
- ▶ Type macros can generate arbitrary amounts of publicly visible defs
- ▶ Enables an astounding multitude of techniques
- ▶ The problem of erasure

Scala in the future: Macro annotations

```
class atomic extends MacroAnnotation {  
  def complete(defn: _) = macro("generate a backing field")  
  def typeCheck(defn: _) = macro("return defn itself")  
}
```

```
@atomic var fld: Int
```

- ▶ Statically-typed analogue of Python's decorators
- ▶ Operates on arbitrary definitions
- ▶ Two-step expansion: macro-level + micro-level

Typechecking disciplines: Strict

```
-| fun pow n = <fn x =>  
    ~(if n = 0 then <1> else <x * (~(pow (n - 1)) x)>>>;  
val pow = fn : int -> <int -> int>  
  
-| val cube = (pow 3);  
val cube = <(fn a => x %* x %* x %* 1)> : <int -> int>  
  
-| (run cube) 5;  
val it = 125 : int
```

- ▶ Each quasiquote is typechecked in isolation
- ▶ All quasiquotes are assigned "code of something" types
- ▶ E.g. right-hand side of `pow` is a code of function from `int` to `int`
- ▶ Hence no pattern matching and no new bindings

Typechecking disciplines: Lenient

```
[| 'a' + True |] -- rejected
```

```
printf :: String -> Expr -- allowed  
$(printf "Error: %s on line %d") "urk" 341
```

```
f :: Q Type -> Q [Dec] -- rejected  
f t = [d | data T = MkT $t; g (MkT x) = x + 1 |]
```

- ▶ Quasiquotes are sanity-checked early, fully typechecked later
- ▶ But require their bindings to be established in advance
- ▶ Not flexible enough, e.g. no splicing into binding positions

Typechecking disciplines: Deferred

```
macro using(name, expr, body) {  
  <[  
    def $name = $expr;  
    try { $body } finally { $name.Dispose() }  
  ]>  
}  
using(db, Database("localhost"), db.LoadData())
```

- ▶ Quasiquotes are not typechecked at all
- ▶ Typechecking only happens after macro expansion
- ▶ This gives ultimate flexibility at the cost of delayed error detection