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 concept?

What are macros?

Macros in programming languages:

- C macros
- Lisp macros
- ...

What is the underlying concept?

Macros realize the notion of textual abstraction:

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

Why macros?

Work with 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)
- ...

Macrology

- Notation
- Bindings
- Typechecking
- Syntax extensibility
- ...

Outline

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
- Impressive power for minimal investments from the compiler

Outline

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

Outline

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 this 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_a1mx = calculate
    it_a1my = temp_a1mx
in if (temp_a1mx /= 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_a1mx = calculate
    it_a1my = temp_a1mx
in if (temp_a1mx /= 0)
   then putStrLn (show it_a1my)
   else error "does not compute"
```

Template Haskell: Summary

- Template Haskell is auto hygienic and referentially transparent
- 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;                          // vanilla 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;                          // vanilla 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;                          // vanilla 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;                          // vanilla color
aif(calculate,
  WriteLine(it),
  throw Exception("does not compute"))

macro aif(cond, then, else_) {               // expansion color
  <[
    def temp = $cond;
    def $("it": usesite) = 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: Hacks don't scale

```
(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: Hacks don't scale

```
(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: Hacks don't scale

```
(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: 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: overall

- There are algorithms that take care of hygiene and referential transparency
- These algorithms can work in automatic mode, but are flexible enough to give the programmer full control
- Like a silver bullet
- Nevertheless sometimes even better solutions come from integration with language features

Outline

scalamacros.org

- Since this semester 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

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]
```

Research proposal

Marry macros and high-level language features:

- Macros + functions $\rightarrow$ programmable inlining, specialization, fusion
- Macros + annotations $\rightarrow$ code contracts
- Macros + path-dependent types $\rightarrow$ controlled effects
- Macros + implicits $\rightarrow$ static verification
- ...