# Metaprogramming with Macros

Eugene Burmako
LAMP, I&C, EPFL

***Abstract*—Macros realize the notion of textual abstraction. Textual abstraction consists of recognizing pieces of text that match a specification and replacing them according to a procedure.**

**In the focus of the study are syntactic macros in lexically scoped programming languages. We identify the problems of *hygiene* and *referential transparency* and describe the solutions employed in Template Haskell [1], Nemerle [2] and Racket [3].**

**We discuss integration of hygienic macros into statically typed languages and propose to improve upon state of the art by providing a type system for syntax templates and uncovering synergies with high-level language features such as path-dependent types and implicits [4].**

***Index Terms*—metaprogramming, macros, quasiquotes, hygiene, referential transparency**

## I. Introduction

PROCEDURAL abstraction is pervasive. Factoring out parameterized fragments of programs into procedures is a conventional best practice.

Modern programming languages integrate the notion of procedures into their semantics. Procedures are viewed as independent programs that can communicate with the main program. As of such they can be manipulated as units, and big procedures can be built from the smaller ones. This is a powerful way to manage complexity of software systems.

Proposal submitted to committee: September 3rd, 2012; Candidacy exam date: September 10th, 2012; Candidacy exam committee: Christoph Koch, Martin Odersky, Viktor Kuncak.



However procedural abstraction is sometimes not enough, because its manifestations are bound by language syntax and it operates within the semantics of the language.

For example, in most programming languages it is impossible to define short-circuiting logical operators as procedures, because procedures are usually not in control of operational semantics. Another example in this vein is a C-like `for` loop, which supports optional prologue that introduces variables visible in its body. Procedures typically cannot abstract over variable bindings, so they cannot express this language construct.

*Textual abstraction* consists of recognizing pieces of text that match a specification and replacing them according to a procedure. Matched fragments are referred to as macro calls or macro applications, and procedures that transform them are dubbed macros or macro transformers. The process of applying macros is called macro expansion [5].

Having means of textual abstraction in their toolbox, programmers can use a multitude of techniques, some of which are:

- reification (providing programs with ways to treat code as data),
- language virtualization (overloading/overriding semantics of the original programming language to enable deep embedding of DSLs),
- programmable optimization (application of optimizations such as inlining or fusion based on knowledge about the program being optimized),
- static verification (using reified representation of the program and, possibly, its contracts defined alongside the program, to verify invariants at compile time),
- algorithmic program construction (generation of code that is tedious to write with the abstractions supported by a programming language).

One possibility to implement a macro system is having it as a standalone tool operating on character streams. This gives rise to lexical macros. Such a design warrants simplicity of the implementation, but undermines robustness, because macros operating on lexical level have no mechanism that prevents generation of syntactically invalid programs.

Another approach would be to integrate a macro expander into the compiler and have macros work with syntax trees, introducing *syntactic macros*. In this model macro application is a node in the program tree, and macro expansion produces a new node that replaces the macro application without distorting the structure of the program.

Problems inherent to syntactic macros can be divided into two categories: inadvertent variable capture and semantically invalid expansions. The rest of the papers dwells upon these challenges.

```
(let ((x 40) (y 2)) (print (+ x y)))

((lambda (x y) (print (+ x y))) (40 2))
```

a) Examples of a macro application and a macro expansion of the `let` macro

```
(defmacro let args
  (cons
    (cons 'lambda
          (cons (map car (car args))
                (cdr args)))
    (map cadr (car args))))
```

b) Creates the result at low level (manipulates S-expressions with standard symbol and list processing functions)

```
(defmacro let (decls body)
 `(
   (lambda
     ,(map car decls)   ;; (x y)
     ,body)             ;; (print (+ x y))
   ,@(map cadr decls))) ;; (40 2)
```

c) Uses quasiquotes [6] as a templating mechanism (backquote introduces a static code template, commas splice dynamic values into the template)

```
(define-syntax let
  (syntax-rules ()
    ((let ((name expr) ...) body ...)
    ((lambda (name ...) body ...) expr ...))))
```

d) Macro-by-example notation [7] (uses a tree matching macro that can extract and reassemble fragments of syntax objects; ellipses capture recurrent parts of the input)

```
(define-syntax (let stx)
  (syntax-parse stx
    ((let ((name:identifier expr:expr) ...) body:expr ...)
     #:fail-when (check-duplicate #'(var ...))
                 "duplicate variable name"
     #'((lambda (name ...) body ...) expr ...))))
```

e) Macro-by-example notation augmented with a syntax specification [8] (colons denote syntax classes, which are first-class and can be built from the ground up).

Figure 1. Assorted implementations of the `let` macro in Lisp dialects

## II. EXAMPLES

`let` is a language construct typical to functional programming. It introduces a scope for a computation and brings temporary variables with provided values into that scope. To implement `let` the compiler might wrap the computation in a lambda abstraction and apply it right away (Figure 1a).

This notion cannot be abstracted procedurally, because the body of the computation typically contains free variables. However textual abstraction fits the bill, because macros can manipulate the program on a level where bindings don't exist and therefore don't impose restrictions. To set up a stage for further chapters, let's implement the `let` macro in Lisp.

The most straightforward solution to the problem is a low-level macro transformer (Figure 1b). It takes an S-expression that represents a macro application, destructures it using standard list manipulation functions, such as `car` and `cdr`, and creates a new S-expression with `cons`. Even in this simple example this notation is very noisy. It's quite difficult to figure out expected shapes of input and output expressions from the imperative algorithm.

Quasiquotes [6] make it possible to reduce obscurity of the macro by providing a domain-specific language for syntax templates (Figure 1c). The quasiquote operator (`` ` ``) demarcates a static template. Quasiquoted code is inserted verbatim into the output (that's why there's no longer need in explicit `cons`'ing). Unquote operators (`,` and `,@`) interrupt a quasiquote, producing "holes" filled in with dynamically calculated data. For example, for `let` we statically know the shape of code to produce (an application of a lambda abstraction) - this makes up the static part of the quasiquote. On the other hand, body and parameters of the lambda as well as the arguments of the application may vary from expansion to expansion - this is the dynamic part.

Another simplification of the macro can be achieved with MBE, the macro-by-example notation [7]. In their seminal work Kohlbecker and Wand came up with a specification of a pattern matcher that matches singular and repetitive parts of S-expressions. Identifiers which appear in the input pattern are treated as pattern variables, ellipses (`...`) used as a last element of a list that contains pattern variables denote repetition and, when nested, can capture lists of arbitrary depth. The revised version of the `let` macro is particularly minimalistic (Figure 1d).

A recent development of the MBE syntax has been proposed by Culpepper and Felleisen [8]. Their refinement addresses the need for principled input validation and error reporting. Indeed, MBE covers the success path, but doesn't help with detecting errors. For example, duplicate identifier names as in `(let ((x 40) (x 2)) (print (+ x y))` will go unnoticed until the compiler gets to the resulting lambda form, which will produce confusing error messages. Authors enhance MBE with both declarative and procedural means of validation (Figure 1e). Colons next to the names of pattern variables denote syntax classes, which put restrictions on the shape of the variables, `#:fail-when` clauses can contain validation code and error messages (this doesn't cover all the capabilities of syntax specifications, please, refer to [8] for details). These validation facilities can be packed into custom syntax classes, which can be built from the ground up.

## References

[1] T. Sheard and S. Peyton Jones, Template meta-programming for Haskell. ACM SIGPLAN Notices, 2002.
[2] K. Skalski, M. Moskal and P. Olszta, Meta-programming in Nemerle. Generative Programming and Component Engineering, 2004.
[3] E. Barzilay, R. Culpepper and M. Flatt, Keeping it Clean with Syntax Parameters. Scheme and Functional Programming Workshop, 2011.
[4] M. Odersky, L. Spoon and B. Venners, Programming in Scala 2nd Edition. Artima, 2010.
[5] E. Kohlbecker, Syntactic Extensions in the Programming Language Lisp. PhD thesis, Indiana University, 1986.
[6] A. Bawden, Quasiquotation in Lisp, Partial Evaluation and SemanticBased Program Manipulation, 1999.
[7] E. Kohlbecker, M. Wand, Macro-by-Example: Deriving Syntactic Transformations from their Specifications, Principles of Programming Languages, 1987.
[8] R. Culpepper, M. Felleisen, Fortifying Macros, International Conference on Functional Programming, 2010.