The History of Macros: A Guided Tour

Stephen Chang

PL, Jr. Seminar

11/15/2010

Scheme
The History of ^ Macros: A Guided Tour

Stephen Chang

PL, Jr. Seminar

11/15/2010

Scheme (mostly) The History of ^ Macros ^: A Guided Tour

Stephen Chang

PL, Jr. Seminar

11/15/2010

Abstraction Mechanisms in Programming Languages

- Functions (Procedural Abstraction)

Abstraction Mechanisms in Programming Languages

- Functions (Procedural Abstraction)
- Structs, Records, Objects, etc. (Data Abstraction)

Abstraction Mechanisms in Programming Languages

- Functions (Procedural Abstraction)
- Structs, Records, Objects, etc. (Data Abstraction)
- Macros (Syntactic Abstraction)

Q: Can't I just use functions to do anything macros can do?

Q: Can't I just use functions to do anything macros can do?

Q: Can't I just use functions to do anything macros can do?

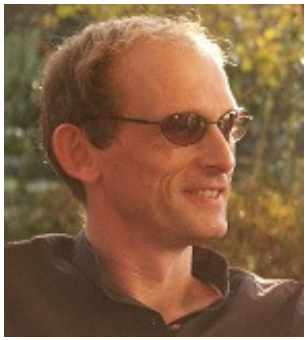
--- NO!

Q: Can't I just use functions to do anything macros can do?

 --- NO!

When To Use a Macro Instead of a Function:

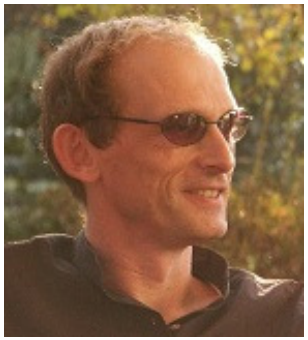
Q: Can't I just use functions to do anything macros can do?

 --- NO!

When To Use a Macro Instead of a Function:

- To use a different order of evaluation

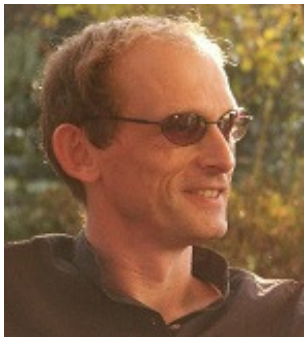
Q: Can't I just use functions to do anything macros can do?

 --- NO!

When To Use a Macro Instead of a Function:

- To use a different order of evaluation
- To introduce new binding constructs

Q: Can't I just use functions to do anything macros can do?

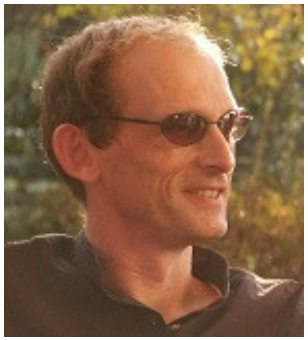


--- NO!

When To Use a Macro Instead of a Function:

- To use a different order of evaluation
- To introduce new binding constructs
- To introduce new syntax

Q: Can't I just use functions to do anything macros can do?

 --- NO!

When To Use a Macro Instead of a Function:

- To use a different order of evaluation
- To introduce new binding constructs
- To introduce new syntax / define new DSLs

Macro Examples: New Syntax

Macro Examples: New Syntax

```
(cond
  ([guard expr]
       ...
   [else else-expr])
```

Macro Examples: New Syntax

```
(cond
  ([guard expr]
       ...
   [else else-expr])
```

≡

```
(if guard
    expr
    (if
      ...
        else-expr))
```

(≡ means "macro definition")

Macro Examples: New Binding Construct

Macro Examples: New Binding Construct

```
(let ([var rhs] ...) body)
```

Macro Examples: New Binding Construct

```
(let ([var rhs] ...) body)
```

≡

```
((λ (var ...) body) rhs ...)
```

Macro Examples: Change Evaluation Order

Macro Examples: Change Evaluation Order

```
(calculate-run-time expr)
```

Macro Examples: Change Evaluation Order

```
(calculate-run-time expr)
```

≡

```
(let ([start-time (current-time)])
  (begin
    expr
    (- (current-time) start-time)))
```

When Not to Use Macros

When Not to Use Macros

Do not use macros for efficiency / inlining!!!

(like C programmers used to do)

Naive Macro Expansion Algorithm

1) find all macro calls in a program and expand (transcription)

2) repeat with expanded program until there are no more macro calls

Naive Macro Expansion Algorithm

1) find all macro calls in a program and expand (transcription)

2) repeat with expanded program until there are no more macro calls

Used in languages like C and Lisp

Naive Macro Expansion Problem: Hygiene

Macro Definition:

```
(or e1 e2)
     ≡
(let ([v e1])
  (if v v e2))
```

Naive Macro Expansion Problem: Hygiene

Macro Definition:

```
(or e1 e2)
     ≡
(let ([v e1])
  (if v v e2))
```

Macro Use:

```
(let ([v true])
  (or false v))
```

Naive Macro Expansion Problem: Hygiene

Macro Definition:

```
(or e1 e2)
     ≡
(let ([v e1])
  (if v v e2))
```

Macro Use:

```
(let ([v true])
  (or false v))
```

macro expands to

```
(let ([v true])
  (let ([v false])
    (if v v v))
```

Naive Macro Expansion Problem: Hygiene

Macro Definition:

```
(or e1 e2)
     ≡
(let ([v e1])
  (if v v e2))
```

Macro Use:

```
(let ([v true])
  (or false v))
```

macro expands to

```
(let ([v true])
  (let ([v false])
    (if v v v))
```

Macro Hygiene: Variable bindings introduced by a macro should not capture variables used at the macro call site.

Naive Macro Expansion Problem: Hygiene

Macro Definition:

```
(or e1 e2)
     ≡
(let ([v e1])
  (if v v e2))
```

Macro Use:

```
(let ([v true])
  (or false v))
```

macro expands to

```
(let ([v true])
  (let ([v false])
    (if v v v))
```

Macro Hygiene: Variable bindings introduced by a macro should ~~not capture variables used at the macro call site.~~
only bind variables introduced by that macro.

Bad Ways To Address Hygiene

- Use obscure variable names in the macro

Bad Ways To Address Hygiene

- Use obscure variable names in the macro
  - `_____macro_variable_____`

Bad Ways To Address Hygiene

- Use obscure variable names in the macro
  - `_____macro_variable_____`
  - `____dont_use_me_as_a_variable___`

Bad Ways To Address Hygiene

- Use obscure variable names in the macro
  - `_____macro_variable_____`
  - `____dont_use_me_as_a_variable___`
- The programmer should just be careful

Bad Ways To Address Hygiene

- Use obscure variable names in the macro
  - `_____macro_variable_____`
  - `____dont_use_me_as_a_variable___`
- The programmer should just be careful

Bigger Problem: Programmer has to deal with this crap

[Kohlbecker, Friedman, Felleisen, Duba - "Hygienic Macro Expansion", 1986 LFP]

[Kohlbecker, Friedman, Felleisen, Duba - "Hygienic Macro Expansion", 1986 LFP]

"... we propose a change to the naive macro expansion algorithm which automatically maintains hygienic conditions during expansion time."

[Kohlbecker, Friedman, Felleisen, Duba - "Hygienic Macro Expansion", 1986 LFP]

"... the task of safely renaming macro-generated identifiers is mechanical. It is essentially an α-conversion which is knowledgeable about the origin of identifiers. ... From the λ-calculus, one knows that if the hygiene condition does not hold, it can be established by an appropriate number of α-conversions. That is also the basis of our solution."

[Kohlbecker, Friedman, Felleisen, Duba - "Hygienic Macro Expansion", 1986 LFP]

"Ideally, α-conversions should be applied with every transformation step, but that is impossible. One cannot know in advance which macro-generated identifier will end up in a binding position. Hence it is a quite natural requirement that one retains the information about the origin of an identifier. To this end, we combine the expansion algorithm with a tracking mechanism."

[Kohlbecker, Friedman, Felleisen, Duba - "Hygienic Macro Expansion", 1986 LFP]

"Tracking is accomplished with a time-stamping scheme. Time-stamps, sometimes called clock values, are simply non-negative integers. The domain of time-stamped variables is isomorphic to the product of identifiers and non-negative integers."

KFFD Algorithm

[Kohlbecker, Friedman, Felleisen, Duba - "Hygienic Macro Expansion", 1986 LFP]

1) stamp all variables in initial program with time-stamp 0

2) do macro expansion

3) increment time-stamp counter and stamp new variables

4) repeat steps 2 and 3 until there are no more macro calls

5) rename time-stamped variables that only differ in their timestamps

6) remove all time-stamps

KFFD Algorithm Example

[Kohlbecker, Friedman, Felleisen, Duba - "Hygienic Macro Expansion", 1986 LFP]

Macro Definition:

```
(or e1 e2)
     ≡
(let ([v e1])
  (if v v e2))
```

KFFD Algorithm Example

[Kohlbecker, Friedman, Felleisen, Duba - "Hygienic Macro Expansion", 1986 LFP]

Macro Definition:

```
(or e1 e2)
     ≡
(let ([v e1])
  (if v v e2))
```

Macro Use:

```
(let ([v true])
  (or false v))
```

KFFD Algorithm Example

[Kohlbecker, Friedman, Felleisen, Duba - "Hygienic Macro Expansion", 1986 LFP]

Macro Definition:

```
(or e1 e2)
     ≡
(let ([v e1])
  (if v v e2))
```

Macro Use:

```
(let ([v true])
  (or false v))
```

macro expands to

```
(let ([v:0 true])
  (let ([v:1 false])
    (if v:1 v:1 v:0)))
```

Lisp `let` Example

`let`

Lisp `let` Example

```
                let

                 ≡
(lambda (s)
  (cons
    (cons 'lambda
          ((cons (map car (cadr s))
                 (cddr s)))
    (map cadr (cadr s))))
```

Lisp `let` Example

```
let
```

≡

```
(lambda (s)
  (cons
    (cons 'lambda
          ((cons (map car (cadr s))
                 (cddr s)))
    (map cadr (cadr s))))
```

≡

```
(defmacro let (decls . body)
  '((lambda ,(map car decls) . ,body)
    . ,(map cadr decls)))
```

[Kohlbecker, Wand - "Macro-by-Example", 1987 POPL]

[Kohlbecker, Wand - "Macro-by-Example", 1987 POPL]

"Even in languages such as Lisp that allow syntactic abstractions, the process of defining them is notoriously difficult and error-prone."

`syntax-rules`

[Kohlbecker, Wand - "Macro-by-Example", 1987 POPL]

`syntax-rules`

[Kohlbecker, Wand - "Macro-by-Example", 1987 POPL]

(predecessor to)

`syntax-rules`

[Kohlbecker, Wand - "Macro-by-Example", 1987 POPL]

(predecessor to)

- pattern matching
- ellipses matches repetitive elements
- multiple cases
- hygiene (using Kohlbecker, et al. 1986)

`syntax-rules`

[Kohlbecker, Wand - "Macro-by-Example", 1987 POPL]

(predecessor to)

- pattern matching
- ellipses matches repetitive elements
- multiple cases
- hygiene (using Kohlbecker, et al. 1986)

Allows macro definitions that look like specifications!

`syntax-rules` examples

Summary So Far

[Kohlbecker, Friedman, Felleisen, Duba - "Hygienic Macro Expansion", 1986 LFP]

[Kohlbecker, Wand - "Macro-by-Example", 1987 POPL]

[Bawden, Rees - "Syntactic Closures", 1988 LFP]

[Bawden, Rees - "Syntactic Closures", 1988 LFP]

""Hygienic macro expansion" (of Kohlbecker, et al.) works by "painting" the entire input expression with some distinctive color before passing it in to the expander. Then the returned replacement expression is examined to find those parts that originated from the input expression; these can be identified by their color. The names in the unpainted text are protected from capture by the painted text, and vice versa."

[Bawden, Rees - "Syntactic Closures", 1988 LFP]

"The painting is done without any understanding of the syntax of the input expression. Paint is applied to expressions, quoted constants, cond-clauses, and the bound variable lists from lambda-expressions. This strikes us as being very undisciplined. We prefer a scheme that is everywhere sensitive to the underlying syntactic and semantic structure of the language. In addition, it is difficult to comprehend how hygienic expansion operates and why it is correct."

[Bawden, Rees - "Syntactic Closures", 1988 LFP]

"The painting is done without any understanding of the syntax of the input expression. Paint is applied to expressions, quoted constants, cond-clauses, and the bound variable lists from lambda-expressions. This strikes us as being very undisciplined. We prefer a scheme that is everywhere sensitive to the underlying syntactic and semantic structure of the language. In addition, it is difficult to comprehend how hygienic expansion operates and why it is correct."

"We feel that syntactic closures solve scoping problems in a natural, straightforward way."

[Bawden, Rees - "Syntactic Closures", 1988 LFP]

"In the same way that closures of lambda-expressions solve scoping problems at run time, we propose to introduce syntactic closures as a way to solve scoping problems at macro-expansion time."

[Bawden, Rees - "Syntactic Closures", 1988 LFP]

"In the same way that closures of lambda-expressions solve scoping problems at run time, we propose to introduce syntactic closures as a way to solve scoping problems at macro-expansion time."

$$\text{syntactic closures} = \text{syntax} + \text{lexical information}$$

Summary So Far

[Kohlbecker, Friedman, Felleisen, Duba - "Hygienic Macro Expansion", 1986 LFP]

[Kohlbecker, Wand - "Macro-by-Example", 1987 POPL]

[Bawden, Rees - "Syntactic Closures", 1988 LFP]

[Clinger, Rees - "Macros That Work", 1991 POPL]

[Clinger, Rees - "Macros That Work", 1991 POPL]

"The problem with syntactic closures is that they are inherently low-level and therefore difficult to use correctly ... It is impossible to construct a pattern-based, automatically hygienic macro system on top of syntactic closures because the pattern interpreter must be able to determine the syntactic role of an identifier (in order to close it in the correct syntactic environment) before macro expansion has made that role apparent."

```
(let ([var rhs] ...) body)
                ≡
((λ (var ...) body) val ...)
```

[Clinger, Rees - "Macros That Work", 1991 POPL]

"Consider the `let` macro. When this macro is used the `rhs` expressions must be closed in the syntactic environment of the use, but the `body` cannot simply be closed in the syntactic environment of the use because its references to `var` must be left free. The pattern interpreter cannot make this distinction unaided until the lambda expression is expanded, and even then it must somehow remember that the bound variable of the lambda expression came from the original source code and must therefore be permitted to capture the references that occur in `body`."

[Clinger, Rees - "Macros That Work", 1991 POPL]

"This paper unifies and extends the competing paradigms of hygienic macro expansion and syntactic closures to obtain an algorithm that combines the benefits of both."

[Clinger, Rees - "Macros That Work", 1991 POPL]

"Unlike previous algorithms, our algorithm runs in linear instead of quadratic time."

[Clinger, Rees - "Macros That Work", 1991 POPL]

"Unlike previous algorithms, our algorithm runs in linear instead of quadratic time."

"The reason that previous algorithms for hygienic macro expansion are quadratic in time is that they expand each use of a macro by performing naive expansion followed by an extra scan of the expanded code to find and paint (i.e. - rename, or time-stamp) the newly introduced identifiers. If macros expand into uses of still other macros with more or less the same actual parameters, which often happens, then large fragments of code may be scanned anew each time a macro is expanded."

[Clinger, Rees - "Macros That Work", 1991 POPL]

"Unlike previous algorithms, our algorithm runs in linear instead of quadratic time."

"The reason that previous algorithms for hygienic macro expansion are quadratic in time is that they expand each use of a macro by performing naive expansion followed by an extra scan of the expanded code to find and paint (i.e. - rename, or time-stamp) the newly introduced identifiers. If macros expand into uses of still other macros with more or less the same actual parameters, which often happens, then large fragments of code may be scanned anew each time a macro is expanded."

"Our algorithm runs in linear time because it finds the newly introduced identifiers by scanning the rewrite rules, and paints these identifiers as they are introduced during macro expansion. The algorithm therefore scans the expanded code but once, for the purpose of completing the recursive expansion of the code tree, just as in the naive macro expansion algorithm."

[Clinger, Rees - "Macros That Work", 1991 POPL]

"Consider an analogy from lambda calculus. In reducing an expression to normal form by textual substitution, it is sometimes necessary to rename variables as part of a beta reduction. It doesn't work to perform all the (naive) beta reductions first, without renaming, and then to perform all the necessary alpha conversions; by then it is too late. Nor does it work to do all the alpha conversions first, because beta reductions introduce new opporunities for name clashes. The renamings must be *interleaved* with the (naive) beta reductions, which is the reason why the notion of substitution required by the non-naive beta rule is so complicated."

[Clinger, Rees - "Macros That Work", 1991 POPL]

"Consider an analogy from lambda calculus. In reducing an expression to normal form by textual substitution, it is sometimes necessary to rename variables as part of a beta reduction. It doesn't work to perform all the (naive) beta reductions first, without renaming, and then to perform all the necessary alpha conversions; by then it is too late. Nor does it work to do all the alpha conversions first, because beta reductions introduce new opporunities for name clashes. The renamings must be *interleaved* with the (naive) beta reductions, which is the reason why the notion of substitution required by the non-naive beta rule is so complicated."

"The same situation holds for macro expansions. It does not work to simply expand all macro calls and then rename variables, nor can the renamings be performed before expansion. The two processes must be interleaved in an appropriate manner. A correct and efficient realization of this interleaving is our primary contribution."

Naive Macro Expansion Problem: Referential Transparency

Macro Definition:

```
(or e1 e2)
     ≡
(let ([v e1])
  (if v v e2))
```

Naive Macro Expansion Problem: Referential Transparency

Macro Definition:

```
(or e1 e2)
     ≡
(let ([v e1])
  (if v v e2))
```

Macro Use:

```
(let ([if and]))
  (or false true)
```

Naive Macro Expansion Problem: Referential Transparency

Macro Definition:

```
(or e1 e2)
     ≡
(let ([v e1])
  (if v v e2))
```

Macro Use:

```
(let ([if and]))
  (or false true)
```

macro expands to

```
(let ([if and])
  (let ([v false])
    (if v v true))
```

[Clinger, Rees - "Macros That Work", 1991 POPL]

"We would like for free variables that occur on the right hand side of a rewriting rule for a macro to be resolved in the lexical environment of the macro **definition** instead of being resolved in the lexical environment of the **use** of the macro."

[Clinger, Rees - "Macros That Work", 1991 POPL]

"We would like for free variables that occur on the right hand side of a rewriting rule for a macro to be resolved in the lexical environment of the macro **definition** instead of being resolved in the lexical environment of the **use** of the macro."

...

"Our algorithm supports referentially transparent macros."

[Clinger, Rees - "Macros That Work", 1991 POPL]

"We would like for free variables that occur on the right hand side of a rewriting rule for a macro to be resolved in the lexical environment of the macro **definition** instead of being resolved in the lexical environment of the **use** of the macro."

...

"Our algorithm supports referentially transparent macros."

(unlike previous algorithms)

[Clinger, Rees - "Macros That Work", 1991 POPL]

"A high-level macro system similar to that described here is currently implemented on top of a compatible low-level system that is not described in this paper."

Summary So Far

[Kohlbecker, Friedman, Felleisen, Duba - "Hygienic Macro Expansion", 1986 LFP]

[Kohlbecker, Wand - "Macro-by-Example", 1987 POPL]

[Bawden, Rees - "Syntactic Closures", 1988 LFP]

[Clinger, Rees - "Macros That Work", 1991 POPL]

[Dybvig, Hieb, Bruggeman - "Syntactic Abstraction in Scheme", 1992 LSC]

[Dybvig, Hieb, Bruggeman - "Syntactic Abstraction in Scheme", 1992 LSC]

"Their system, however, allows macros to be written only in a restricted high-level specification language in which it is easy to determine where new identifiers will appear in the output of a macro. Since some macros cannot be expressed using this language, they have developed a low-level interface that requires new identifiers to be marked explicitly."

[Dybvig, Hieb, Bruggeman - "Syntactic Abstraction in Scheme", 1992 LSC]

"Lisp macro systems cannot track source code through the macro-expansion process. Reliable correlation of source code and macro-expanded code is necessary if the compiler, run-time system, and debugger are to communicate with the program in terms of the original source program."

`syntax-case`

[Dybvig, Hieb, Bruggeman - "Syntactic Abstraction in Scheme", 1992 LSC]

`syntax-case`

[Dybvig, Hieb, Bruggeman - "Syntactic Abstraction in Scheme", 1992 LSC]

- combines high-level and low-level systems
- (can write unhygienic macros)
- fenders on each case
- referential transparency
- source code matched to expanded code

`syntax-case`

[Dybvig, Hieb, Bruggeman - "Syntactic Abstraction in Scheme", 1992 LSC]

- combines high-level and low-level systems
- (can write unhygienic macros)
- fenders on each case
- referential transparency
- source code matched to expanded code

syntax object = syntax + lexical information + source locations

`syntax-case` examples

Summary So Far

[Kohlbecker, Friedman, Felleisen, Duba - "Hygienic Macro Expansion", 1986 LFP]

[Kohlbecker, Wand - "Macro-by-Example", 1987 POPL]

[Bawden, Rees - "Syntactic Closures", 1988 LFP]

[Clinger, Rees - "Macros That Work", 1991 POPL]

[Dybvig, Hieb, Bruggeman - "Syntactic Abstraction in Scheme", 1992 LSC]

[Clinger, Rees - "Macros That Work", 1991 POPL]

"One project we intend to pursue is to integrate our algorithm with a module system for Scheme."

[Flatt - "Composable and Compilable Macros", 2002 ICFP]

[Flatt - "Composable and Compilable Macros", 2002 ICFP]

"In the Lisp and Scheme tradition, where macros are themselves defined in a macro-extensible language, extensions can be stacked in a "language tower." Each extension of the language can be used in implementing the next extension. ... Advances in macro technology have simplified the creation of individual blocks for a tower, but they have not delivered a reliable mortar for assembling the blocks.

[Flatt - "Composable and Compilable Macros", 2002 ICFP]

"Suppose `P.scm` is implemented in an extension of Scheme, *E*, where *E* is implemented by `E.scm` directly in Scheme. A typical load sequence for *P* is

```
(load "E.scm")

(load "P.scm")
```

The above statements might be placed in a file `loadP.scm`, which can then be submitted to be a Scheme interpreter to execute `P.scm` successfully."

[Flatt - "Composable and Compilable Macros", 2002 ICFP]

"The problem starts when the programmer tries to compile the program for later execution. Supplying `loadP.scm` to the compiler is useless, because the result is simply the compiled form of two `load` statements. A full compiler will be needed at run-time when `P.scm` is actually loaded."

[Flatt - "Composable and Compilable Macros", 2002 ICFP]

"The problem starts when the programmer tries to compile the program for later execution. Supplying `loadP.scm` to the compiler is useless, because the result is simply the compiled form of two `load` statements. A full compiler will be needed at run-time when `P.scm` is actually loaded."

"The problem is that the compile-time code in `E.scm` is not distinguished in any way from the run-time code in `P.scm`, and the run-time `load` operation is abused as a configuration-time operation."

Phase Examples in Racket

Summary So Far

[Kohlbecker, Friedman, Felleisen, Duba - "Hygienic Macro Expansion", 1986 LFP]

[Kohlbecker, Wand - "Macro-by-Example", 1987 POPL]

[Bawden, Rees - "Syntactic Closures", 1988 LFP]

[Clinger, Rees - "Macros That Work", 1991 POPL]

[Dybvig, Hieb, Bruggeman - "Syntactic Abstraction in Scheme", 1992 LSC]

[Flatt - "Composable and Compilable Macros", 2002 ICFP]

[Culpepper - "Fortifying Macros", 2010 ICFP]

[Culpepper - "Fortifying Macros", 2010 ICFP]

"Without validation, macros aren't true abstractions."

[Culpepper - "Fortifying Macros", 2010 ICFP]

"Without validation, macros aren't true abstractions."

"Existing systems make it surprisingly difficult to produce easy-to-understand macros that properly validate their syntax."

`syntax-case` with guards

```
(define-syntax (checked-let1 stx)
  (syntax-case stx ()
    [(_ ([var rhs] ...) body)
     (and (andmap identifier? (syntax->list #'(var ...)))
          (not (check-duplicate-identifier #'(var ...))))
     #'((λ (var ...) body) rhs ...)]))
```

[Culpepper - "Fortifying Macros", 2010 ICFP]

"Guard expressions suffice to prevent macros from accepting invalid syntax, but they suffer from two flaws."

[Culpepper - "Fortifying Macros", 2010 ICFP]

"Guard expressions suffice to prevent macros from accepting invalid syntax, but they suffer from two flaws."

1) "Since guard expressions are separated from transformation expressions, work needed both for validation and transformation must be performed twice and code is often duplicated."

[Culpepper - "Fortifying Macros", 2010 ICFP]

"Guard expressions suffice to prevent macros from accepting invalid syntax, but they suffer from two flaws."

1) "Since guard expressions are separated from transformation expressions, work needed both for validation and transformation must be performed twice and code is often duplicated."

2) "Guards do not explain why the syntax was invalid. That is, they only control matching; they do not track causes of failure."

`syntax-case` with guards + error messages

```
(define-syntax (checked-let2 stx)
  (syntax-case stx ()
    [(_ ([var rhs] ...) body)
(begin
  (for-each
   (λ (x)
     (unless (identifier? x)
       (raise-syntax-error
        'not-identifier "expected identifier" stx x)))
   (syntax->list #'(var ...)))
  (let ([dup (check-duplicate-identifier #'(var ...))])
   (when dup
     (raise-syntax-error
      'duplicate-var "duplicate variable name" stx dup))))
     #'((λ (var ...) body) rhs ...)])
```

[Culpepper - "Fortifying Macros", 2010 ICFP]

`syntax-parse`

[Culpepper - "Fortifying Macros", 2010 ICFP]

- pattern variables annotated with syntax class
- define new syntax classes
- better error reporting

[Culpepper - "Fortifying Macros", 2010 ICFP]

```
(define-syntax (checked-let3 stx)
  (syntax-parse stx
    [(_ ([var:identifier rhs:expr] ...) body:expr)
     #:fail-when (check-duplicate-identifier #'(var ...))
                 "duplicate variable name"
     #'((λ (var ...) body) rhs ...)]))
```

syntax-class

[Culpepper - "Fortifying Macros", 2010 ICFP]

```
(define-syntax-class binding
  #:description "binding pair"
  (pattern [var:identifier rhs:expr]))
```

[Culpepper - "Fortifying Macros", 2010 ICFP]

```
(define-syntax (checked-let4 stx)
  (syntax-parse stx
    [(_ ([b:binding ...]) body:expr)
     #:fail-when (check-duplicate-identifier #'(b.var ...))
                 "duplicate variable name"
     #'((λ (b.var ...) body) b.rhs ...)])
```

distinct-bindings

[Culpepper - "Fortifying Macros", 2010 ICFP]

```
(define-syntax-class distinct-bindings
  #:description "sequence of binding pairs"
  (pattern (b:binding ...)
           #:fail-when
             (check-duplicate-identifier #'(var ...))
             "duplicate variable name"
           #:with (var ...) #'(b.var ...)
           #:with (rhs ...) #'(b.rhs ...)
```

[Culpepper - "Fortifying Macros", 2010 ICFP]

```
(define-syntax (checked-let5 stx)
  (syntax-parse stx
    [(_ bs:distinct-bindings body:expr)
     #'((λ (bs.var ...) body) bs.rhs ...)])
```

Final Summary

[Kohlbecker, Friedman, Felleisen, Duba - "Hygienic Macro Expansion", 1986 LFP]

[Kohlbecker, Wand - "Macro-by-Example", 1987 POPL]

[Bawden, Rees - "Syntactic Closures", 1988 LFP]

[Clinger, Rees - "Macros That Work", 1991 POPL]

[Dybvig, Hieb, Bruggeman - "Syntactic Abstraction in Scheme", 1992 LSC]

[Flatt - "Composable and Compilable Macros", 2002 ICFP]

[Culpepper - "Fortifying Macros", 2010 ICFP]

Additional Reading

[Culpepper - "Taming Macros", 2004 GPCE]

[Herman - "A Theory of Typed Hygienic Macros", 2010 NEU Dissertation]