Macros that Compose: Systematic Macro Programming

Oleg Kiselyov
FNMOC
oleg@okmij.org

Generative Programming and Component Engineering: ACM SIGPLAN/SIGSOFT Conference, GPCE 2002. Pittsburgh, PA, USA, October 6-8, 2002.

`http://pobox.com/~oleg/ftp/Scheme/macros.html`

Macros that Compose: Systematic Macro Programming

What are macros?

- Syntactic extensions
- Abstract or concrete syntax transformers
- Head-first (normal order) re-writing systems

This talk is about macros or pre-processors. These are facilities to add a new kind of, well, *syntax* to the core language. We will be talking about systems that recursively re-write a phrase in an extended language into a sentence in the core language. So, macro-systems are in general source-to-source transformers, of either the concrete or the tokenized source.

Macro-phrases, or macro-applications, are typically introduced by a head word – a keyword. When a macro processor encounters a keyword, it invokes the corresponding transformer, passes to it the macro-phrase as it is, and re-processes the result. Because the rest of a macro phrase is interpreted solely by a syntactic transformer, the phrase may contain constructs that appear invalid or ill-formed in the core language. This fact makes macros powerful tools for the extension of language syntax.

Examples of macro systems

- `#define`
- Lisp macros
- PL/1 macros
- *Scheme syntax-rewriting-rules*
- Camlp4 quotations
- MacroML
- Tcl
- `/bin/sh`

Lisp macros are probably the most well-known macro facility. Scheme has a high-level tokenized source transformer guided by so-called syntax-rules. MacroML extension for ML and Camlp4 pre-processor for OCaml are examples of macro facilities for modern functional languages.

Most of the examples in this talk will be using syntax-rules macros of Scheme. We chose Scheme macros because they are well-designed, powerful and mature. Besides, we have to write examples in some language. Most of the results and conclusions of the paper *will apply to several of these systems as well*.

Why Macros?

- Conditional compilation and controlled in-lining
- Control: affecting the evaluation order
- Binding
- Domain-specific, little languages

What makes macros worth talking about? We use macros for conditional compilation and inlining – not only in C but in Scheme, OCaml and Haskell, too. We write macros for convenient switching, looping, and delaying forms – basically to add a degree of laziness to a strict language. Macros can introduce custom binding forms, and, in the best application, can implement domain-specific notations.

Can higher-order functions supplant macros?

- Control macros: **Yes**
- Macros for second-class objects: **No**

Even Haskell needs macros:

- Expression substitution
- Adding whole function definitions
- Adding types or portions of types
- Adding export/import list entries
- Token-pasting for building identifier names

[Keith Wansbrough (1999): Macros and Preprocessing in Haskell.]

One sometimes hears that higher-order functions (and related non-strictness) make macros unnecessary. For example, In Haskell, `if` is a regular function. However, every language with more syntax than lambda-calculus has phrases that are not expressions. Examples of such second-class forms are: type, module, fixity and other *declarations*; binding forms; statements. Only macros can expand into a second-class object. The result of a function is limited to an expression or a value.

Do we really want to manipulate second-order forms? The developers of a Glasgow Haskell Compiler seem to think so. The compiler is the biggest Haskell application. It includes a notable amount of CPP macros. The paper "Keith Wansbrough (1999). Macros and Preprocessing in Haskell. Unpublished. http://www.cl.cam.ac.uk/~kw217/research/misc/hspp-hw99.ps.gz" documents all such uses, and argues that macros should be included into language standards.

The best case for macros

A program configurator

```
(SSAX:make-parser
 NEW-LEVEL-SEED
 (lambda (elem-gi attributes namespaces
                  expected-content seed)
   '())
 UNDECL-ROOT
 (lambda (elem-gi seed)
   (values #f '() namespaces seed))
 ...
)
```

The best application of macros is adding a domain-specific notation to the language, overlaying a little language on the top of a general-purpose one. John Ousterhout and Paul Graham have made compelling arguments for such embeddings. A compelling example of macros – close to the topic of this conference – is a program configurator. I mean a tool to assemble an application out of components. On this slide is an example of instantiating a fast and extensive XML parser in Scheme {SSAX}. The parser is actually a toolkit of parsers and lexers of various sorts, to be combined by the user. This macro {SSAX:make-parser} assembles the parser given particular user callbacks. The macro *integrates* the callbacks, so the resulting parser is fast.

I should note that some of the call-back interfaces are overlapping. Therefore, regular module facilities won't suffice.The make-parser *macro* can know which of the overlapping interfaces is being instantiated, and therefore, can configure the fast or a slow path appropriately.

BSD kernels, Apache modules show more examples of macros as configuration tools.

Dangers of macros

- The expansion code may be mis-formed
- The expansion code may be mis-typed
  →MacroML
- Subtle semantic errors
  →CPP.info: Macro Pitfalls
- Non-obvious interaction of modules and macros
  →Matthew Flatt's paper @ ICFP02
- Hygiene
  →Scheme 2002 workshop
- Macros are too complex
  →

Macros however pose well-known dangers. A carelessly written macro may expand in the code that cannot be typed or even parsed. This is especially easy to do in C. Macros may bind "temporary" variables whose names clash with the names of user variables. The latter problem is called the lack of hygiene. The references {on the slide} indicate that these problem have been or are being addressed. There have been several talks at this PLI specifically about developing macro systems that assuredly behave.

Finally, macros are just too complex. They are difficult to develop and test, which leads to many errors. It takes a degree in macrology to write even a moderately complex macro – and it takes even more advanced degree to understand the macro. That is the problem we are addressing in this talk.

Macro (Mis)Composition

```
#ifdef ...
#define MAXLIMIT 10
#ifdef ...
#define MAXLIMIT 20
...

assert (i<MAXLIMIT);

==>
assertion "i<MAXLIMIT" failed:
    file "a.c", line 7
```

The reason macros are so difficult to comprehend is that they in general, do not compose. The problem is quite frequent – so frequent that we might even got used to it. For example, let's take a macro assert(), which everybody knows and uses. The macro evaluates its argument expression, and if the result is FALSE, it prints the message, quoting the expression that caused the failure.Let us consider this assert statement, where MAXLIMIT is as nullary macro that expands into a integer, depending on a blood-pressure-raising tangle of conditions. Everybody have seen such a mess. It's very difficult to know what the MAXLIMIT is unless we just print it out. Should this assertion fail, we see the the following message. It indicates that i is above MAXLIMIT. It would be helpful to know what was the value of MAXLIMIT at this point – but the message does not say that. Indeed, we see the name of the macro MAXLIMIT rather than the result of its expansion. So, the complete composition of two macros, ASSERT and MAXLIMIT, essentially failed.

Overview

- Macros: the good, the bad, and the mis-composition
- CPS macros always compose!
- Macro-lambda
- Practical methodology of building complex macros *systematically*
  - Example of writing a macro *easily*
  - Automatic translation:
    Scheme code $\rightarrow$ syntax-rule macro

So far, we have seen the good, the bad, and the miscomposition. The latter is quite disconcerting, because it prevents us from building complex macros by combining simpler ones. Miscomposition breaks the fundamental Software Engineering principle of modularity. Because this is a very important property to lose, we will elaborate on it further.

We will see however that macros written in a continuation-passing style (CPS) always compose. Making CPS macros practical still requires an encoding for macro-continuations. We have found this missing piece, with an insight for anonymous macro-level abstractions.

CPS macros with lambda-abstractions lead to a general practical methodology of building macros *systematically*. Macro programming becomes no more complex than regular programming. We present the methodology and give an illustration of how to write a macro easily. In the specific case of Scheme macros, we give a stronger result: developing syntax-rules macros by an automated translation from the corresponding Scheme procedures. We can literally write macros just as we write regular procedures.

Mis-composition - 1

Native `let`

```
(let (( i  (+ 1 4) )) (+ i 37))
        var   init        body
```

Infix `let`

```
(define-syntax lets
  (syntax-rules (<- in)
    ((lets var <- init in body)

     (let ((var init)) body)
)))

(lets i <- (+ 1 4)
      in (+ i 37))
```

As I said before, we will be using Scheme and its high-level macro system for our examples. Most of the results will apply to other macro systems.

As many well-designed languages, Scheme has a binding construct named 'let'. In its simplest form, it looks as follows. The value of the let form is the value of the body in an environment amended by the binding of variable var to the value of the initializing expression init.

The Scheme USENET newsgroup bears witness of many people who have seen the light and come to Scheme. Let us imagine one such person, who has not yet fully recovered from the infix addiction, and wishes for a let form that he is used to in the other language. He can easily extend Scheme with such an infix let form.

The macro facility of Scheme can be called re-writing-by-example. The programmer specifies examples of the code before and after the re-writing. In our example, we wish to rewrite this form into that. Therefore, this expression expands into that and predictably gives the answer to everything.

BTW, this is the case of using a macro to introduce a new, more palatable, domain-specific, notation. The example is intentionally simple. In the pattern, var, init and the body are pattern variables, and the arrow and the word `in` are literals. That's why they are mentioned here {`syntax-rules (<- in)`}

Mis-composition - 2

Functional composition

$$\texttt{f(id(x),y,z)} \equiv \texttt{f(x,y,z)}$$

The identity macro

```
(define-syntax id
  (syntax-rules ()
    ((id x) x)))
```

The simplest case of composition is the composition with the identity function, like here and the same for the other arguments. We can easily write an identity *macro*, a macro that expands into its argument {point to the 'id' syntax rule}. Let us see if the same equality holds for macros.

Mis-composition - 3

-
  ```
  (lets i <- (id (+ 1 4)) in (+ i 37))
  ; ==> 42
  ```

-
  ```
  (lets i <- (+ 1 4) in (id (+ i 37)))
  ; ==> 42
  ```

-
  ```
  (lets (id i) <- (+ 1 4) in (+ i 37))
  ```

  *** ERROR:bigloo:rename-vars: Illegal variable – (id i)

-
  ```
  (lets i (id <-) (+ 1 4)
          in (+ i 37)))
  ```

  *** ERROR:bigloo:expand-syntax: Use of macro does not match definition – `((lets i (id <-) (+ 1 4) in (+ i 37)))`

[Cover the slide]

Let us first wrap the initialization expression into the id macro. Well, the composition works, the evaluation result is unchanged. We now wrap the body – again, it composes. Now we try the composition on the first argument. Oops, what happened here {third case}? And here {fourth case on the slide}?

This is easy to explain if we see the expansion of these macros

Mis-composition - 4

- `(lets i <- (`**id**` (+ 1 4)) in (+ i 37))`
  `→(let ((i `**(id (+ 1 4))**`)) (+ i 37))`

- `(lets i <- (+ 1 4) in (`**id**` (+ i 37)))`
  `→(let ((i  (+ 1 4))) `**(id (+ i 37))**`)`

- `(lets (`**id**` i) <- (+ 1 4) in (+ i 37))`
  `→(let ((`**(id i)**`  (+ 1 4))) (+ i 37))`

- `(lets i (`**id**` <-) (+ 1 4)`
  `      in (+ i 37)))`

  ```
  cf.
    (syntax-rules (<- in)
      ((lets var <- init in body)
  ```

In this case, the argument of the macro `lets` ends up in a position of an initializing *expression* of the native let form. A macro expander is designed to systematically re-write expressions into their normal forms. Therefore, after `lets` is expanded, the macro expander checks the resulting *expressions* for macro phrases. Here {the first case}, it finds one, and expands it. Here {the third case}, however the argument of lets, `(id i)`, appears where a variable is expected. The macro-expander knows that `let` is a special form, and some positions are not for expressions. Therefore `(id i)` just stays here. When the core language compiler gets to that expression, it finds something other than an identifier in the place of a variable to bind. The compiler complains.

We say that the position of a binding variable is a *special class position* in that phrases in this position are not parsed as expressions, and therefore, are not subject to macro-expansion. A phrase under a quote is another example of a special position phrase.

Here, in the fourth example, the situation is different. This argument is matched against a literal, an arrow symbol – and here the match certainly fails. A macro-argument to be deconstructed inside the macro also occupies a special class position.

Macros that place their arguments in special class positions constitute the most compelling class of macros. These macros build bindings, type and module declarations and other second-class phrases. Higher-order functions cannot do that. Such macros are also non-compositional.

In these two cases the problem could have been avoided if the macro `lets` had "evaluated" {(or normalized, to be precise)} its arguments before dealing with them further. Syntax-rules macros however cannot invoke the macro-expander recursively – or any other function, for that matter. Doing so would have made efficient hygienic macro-expansion too difficult or impossible.Unlike a function, a head-first normal-order re-writing rule can force evaluation of an expression only by returning it. However, by doing so, the original rule loses the control, so to speak. If `lets` expands into the form `(id i)`, how would it get back the result of its expansion?

Why mis-composition matters

- Embedding a DSL requires sophisticated macros

- Divide-and-conquer helps

- Higher-order combinators help

Without composition, macros tend to be monolithic, highly recursive, ad hoc, and requiring a revelation to write, and the equal revelation to understand

{This talk however deals with a software-engineering aspect of writing macros.} Embedding a domain-specific notation often requires sophisticated macros. A complex artifact is easier to develop if it can be composed of smaller, separately designed and tested parts. If we do not have composition, the familiar idioms of a function call – let alone higher-order combinators such as map or fold – do not easily apply. Therefore, macros tend to be monolithic, highly recursive, ad hoc, and requiring a revelation to write, and the equal revelation to understand. Many examples of syntax-rules demonstrate that we have to resort to hacks even in simple cases.

Macro-lambda and Macro-apply

```
(??!lambda (bound-var ...)
       (body (??! bound-var) ...))
```

```
(??!apply (??!lambda (bound-var ...) body)
          arg ...)
==>
body[arg/(??! bound-var) ...]
```

We shall now show how to construct macros that are always compositional – even if they place their arguments in special positions. In some circumstances, we can compose with 'legacy' macros written without following our methodology. As a side effect, the technique makes syntax-rule macros functional, restores the modularity principle, and even makes possible higher-order combinators.

The first, novel ingredient to our macro programming technique is a notation for a *first-class parameterized future macro-expansion action*, or macro-lambda for short. Here `bound-var` is a variable, `body` is an expression that may contain these forms $\{$`(??! bound-var)`$\}$, and `??!lambda` is just a symbol. Although the `??!lambda` form may look like a macro-application, it is not. Our macro-level abstractions are not macros themselves, but they are first class.

The `??!lambda`-form is interpreted by a macro `??!apply`. To be more precise, we specify that the following phrase expands into the `body`, with all non-shadowed instances of `(??! bound-var)` replaced by `arg`. In Scheme, question and exclamation marks are considered ordinary characters. Hence `??!lambda`, `??!apply` and `??!` are ordinary symbols – albeit oddly looking, on purpose.

An implementation of ??!apply that satisfies this specification is in the paper. Conceptually, it is trivial: a mere substitutor.

Macro *Composition*

A CPS macro

```
(define-syntax cps-macro
  (syntax-rules ()
    ((_ args k) (??!apply k result))
    ((_ args k) (cps-macro continuation))))
```

How `foo` can invoke `(bar bar-args)`

```
(define-syntax foo
  (syntax-rules ()
    ((_ foo-args k)
     (bar bar-args
          (??!lambda (result)
            (continuation ...
              (??! result) ... k))))))
```

Continuation-passing-style (CPS) macros first introduced by Friedman and Hilsdale are the second component of our methodology. Our CPS macros must receive a continuation argument or arguments, and must expand into an application of `??!apply`. Alternatively, a CPS macro may expand into an application of another CPS macro, whose continuation argument is typically encoded with a macro-lambda. In particular, if a macro `foo` wants to 'invoke' a macro `bar` on an argument `bar-args`, `foo` should expand into this form $\{$`(bar args (??!lambda (res) continuation))`$\}$. Here the continuation argument of `bar` includes `(??! result)` forms and thus encodes what needs to be done with the `bar`'s result.

CPS macros

```
(define-syntax id-cps
  (syntax-rules ()
    ((_ x k)
     (??!apply k x))))


(id-cps i
  (??!lambda (var)
   (id-cps <-
    (??!lambda (arrow)
     (id-cps (+ 1 4)
       (??!lambda (init)
          (lets (??! var) (??! arrow)
                (??! init) in (+ i 37))))))))
```

This technique easily solves the problem of composing macros `id` and `lets`. We will first re-write the macro `id` in CPS. Compared to the macro `id` we saw earlier, `id-cps` receives a continuation argument, and expands into a code that passes the argument x to that continuation. The composition of `id-cps` with `lets` will use a macro-level abstraction to encode the continuation. This macro expands without errors, to the same let form, and evaluates correctly to the universal answer. We can insert forms in regular positions {(??! init)}, and in special positions, as in here {(??! var)} and here {(??! arrow)}.

Moreover, we can show that `id-cps`, in contrast to the macro `id`, *always* composes with any macro. The paper makes a brief formal argument. We suitably abstract the composition problem to that in lambda-calculus. Macro-expansion translates to a call-by-name evaluation. The paper proves that CPS terms always compose, both in call-by-name and call-by-value calculi.

More complex example of a macro composition

```
(define-syntax ?plc-fact
  (syntax-rules ()
    ((_ co k)
     (?plc-zero? co
       (??!lambda (c) (?plc-succ (??! c) k))
       (??!lambda (c) ; on c being > 0
          (?plc-pred (??! c) ; the predecessor
             (??!lambda (c-1)
               (?plc-fact (??! c-1)
                  (??!lambda (c-1-fact)
                     (?plc-mul (??! c)
                               (??! c-1-fact)
                               k))))))))))
```

We show a more elaborate example of developing complex macros by macro composition. It is a compile-time implementation of a factorial over Peano-Church numerals. No computer science paper is complete without working out the factorial. This fragment is quoted here to illustrate modularity: the factorial macro is built by composition of separately defined and tested arithmetic (`?plc-pred`, `?plc-mul`) and comparison macros. Selection forms such as `?plc-zero?` take several continuation arguments but continue only one.

Systematic development of a complex DSL macro

delete-assoc, a part of the SSAX:make-parser

`delete-assoc` deletes an association with the name `tag` from `alist`, a list of `(name . value)` pairs. We return the list of the remaining associations. `tag` not found => error

```
(define (delete-assoc alist tag)
  (let loop ((alist alist) (scanned '()))
    (cond
     ((null? alist)
      (error "Unknown callback-tag: " tag))
     ((eq? tag (caar alist))
      (append scanned (cdr alist)))
     (else
      (loop (cdr alist)
            (cons (car alist) scanned))))))
```

The previous macro looked rather similar to a regular Scheme procedure written in CPS. This observation raises a question if syntax-rule macros can be written systematically – by translating from the corresponding Scheme functions. The answer is Yes. The paper gives a detailed example, incidentally, with a higher-order combinator map. We will talk about a different example, taken from the XML parser configurator macro. To remind, the configurator macro takes a list of call-back procedures. Each procedure is preceded by an identifying tag. The configurator macro needs to merge the user-specified list with the default list. The default list is an associative list, a list of tag-value pairs. One part of the merge operation is removing an association from the list. This operation can be described by the following procedure. It is easy to see what is going on: a mere list traversal. We check if the list is empty and if the association at the head of the list has the tag of interest. Otherwise, we check the next association. The slide shows the simplest form of this procedure. It is a regular procedure, which is easy to design and test.

delete-assoc-cps

```
(define (delete-assoc-cps alist-orig tag ktop)
  (letrec ((loop
    (lambda (alist scanned k)
      (ifnull? alist
        (lambda (_) (error "Unknown callback-tag
        (lambda (_) (caar-cps alist
          (lambda (elem)
            (ifeq? tag elem (lambda (_)
              (cdr-cps alist (lambda (rest)
                (append-cps scanned rest
                 k))))
              (lambda (_) (car-cps alist
                (lambda (head) (cdr-cps alist
                  (lambda (tail)
                    (cons-cps head scanned
                      (lambda (new-scanned)
                        (loop tail
                              new-scanned k)))
                )))))))))))))
    (loop alist-orig '() ktop)))
```

The first step is re-writing the procedure in the continuation-passing-style. Yes, it's not pretty: CPS code rarely is. {We assume CPS versions of basic primitives such as car and cdr, and switching primitives such as ifnull? and ifeq?. The latter takes two argument and two continuations. If the arguments are equivalent, the procedure continues to the first continuation. Otherwise, the second, on-false, continuation is followed.}

This code is more convoluted. It's harder to see what's going on. This is still a regular Scheme procedure, which is relatively easy to test. {The other consolation is that a transformation from the previous procedure is mechanical.}

?delete-assoc

```
(define-syntax ?delete-assoc
 (syntax-rules () ((_ alist-orig tag ktop)
  (letrec-syntax
   ((loop (syntax-rules ()((_ alist scanned k)
     (?ifnull? alist
       (??!lambda (_) (error "Unknown callback-t
       (??!lambda (_)  ; alist is non-empty
         (?caar alist (??!lambda (elem)
           (?ifeq? tag (??! elem)
             (??!lambda (_)
               (?cdr alist (??!lambda (rest)
                 (?append scanned (??! rest)
                   k))))
             (??!lambda (_) (?car alist
               (??!lambda (head) (?cdr alist
                 (??!lambda (tail)
                   (?cons (??! head) scanned
                     (??!lambda (new-scanned)
                       (loop (??! tail)
                             (??! new-scanned)
                            k))))))))))))))))
  (loop alist-orig () ktop)))))
```

On this slide is the result of translating the CPS procedure into a CPS macro. To distinguish CPS macros, we begin their names with the question mark. It is just a notational convention. The question mark has no syntactic significance. As we see the *macro* code looks almost the same as the CPS *procedural* code, modulo question marks and occasional syntax-rules. This is the only sign we are dealing with a macro here.

Normally, syntax-rule pattern-based macros look *nothing* like regular Scheme procedures.

Using and beautifying the CPS macro

```
(?delete-assoc
  ((NEW-LEVEL-SEED . nls-proc)
   (FINISH-ELEMENT . fe-proc)
   (UNDECL-ROOT . ur-proc))
  FINISH-ELEMENT
 (??!lambda (result) (display '(??! result))))
; ==> ((NEW-LEVEL-SEED . nls-proc)
       (UNDECL-ROOT . ur-proc))

(define-syntax _delete-assoc
  (syntax-rules ()
    ((_ alist-orig tag)
     (?delete-assoc alist-orig tag
       (??!lambda (result) (??! result))))))
```

Here is an example of using the designed macro, to delete an association with this tag from this list. It works. It is a modular macro and can be freely composed with other CPS macros. It is a syntax-rule, hence, a hygienic macro.

It is, however, ungainly, because of the continuation argument here {(??!lambda (result)...)}. The ease of use is an important issue for macros, since making the code look pretty is the principal reason for their existence. Therefore, as the last step we wrap `?delete-assoc` into a non-CPS macro

The generic nature of the wrapper is noteworthy. We merely partially apply the macro to an identity continuation.

Finally, we saw that transformations steps from a procedure to a macro are rather regular. Can we do them automatically? The answer is yes.

Scheme -to- Syntax-rule compiler

```
(define-syntax ?delete-assoc
  (syntax-rules () ((_ _?alist _?tag _?kg1029)
    (letrec-syntax ((?loop (syntax-rules ()
      ((_ _?alistg1031 _?scannedg1032 _?kg1030)
       (?ifnull? _?alistg1031
         (??!lambda (g1033)
           (??!apply _?kg1030 (error tag)))
         (??!lambda (g1034) (?car _?alistg1031
           (??!lambda (g1043) (?car (??! g1043)
             (??!lambda (g1042)
               (?eq? _?tag (??! g1042)
                 (??!lambda (g1035)
                  (?iftrue? (??! g1035)
                    (??!lambda (g1036) ...)
                    (??!lambda (g1038)
                      (?cdr _?alistg1031
                        (??!lambda (g1039)
                         (?car _?alistg1031
                           (??!lambda (g1041)
                      ... ))))))))))))))))))
      (?loop _?alist () _?kg1029)))))
```

delete-assoc procedure: the translation source

```
(define (delete-assoc alist tag)
  (let loop ((alist alist) (scanned '()))
    (cond
     ((null? alist)
      (error "Unknown callback-tag: " tag))
     ((eq? tag (caar alist))
      (append scanned (cdr alist)))
     (else
      (loop (cdr alist)
            (cons (car alist) scanned))))))
```

That is exactly what Scheme-to-syntax-rules compiler does. The compiler does both stages – CPS and macro transforms – in one pass. The paper has a link to the full source code of the compiler.

This slide {(define-syntax ?delete-assoc...)} contains a slightly abbreviated code, produced by the compiler from the original simple procedure, here {delete-assoc}. The code is rather close to the one we built by hand. I agree, otherwise the code is a mess. *But it doesn't matter!* We don't have to look at the code. We don't have to know how it works. If the original procedure was correct, the macro will be correct too, and will work without fail. This is the best part of the translation. We don't need a degree in macrology any more to write sophisticated macros. We write pure functional *procedures* that transform S-expressions in the desired way. We test them using standard Scheme facilities. We compile the procedures into a macro, and it works. Furthermore, because the resulting macros are compositional, we enjoy a separate compilation. We can build libraries of transformation procedures and the corresponding macros.

Conclusions

- Special-position macros do not compose, and it is bad
  $\rightarrow$cannot divide-and-conquer

- Macro programming *can* be made systematic and applicative: CPS + macro-lambda

- Syntax-rules as *object code*

We have demonstrated that macro programming is in general non-functional.The fundamental engineering principle of modular design therefore does not generally apply to macros.

We have presented a methodology that makes macro programming applicative and systematic. The methodology is centered on writing macros in continuation-passing style and encoding continuations in macro-lambda abstractions. Therefore, complex macros can be constructed by combining separately developed and tested components, by composition and higher-order operators.

Scheme syntax-rule macros specifically admit a stronger result: syntax-rule macros can be systematically developed as regular, tail-recursive Scheme procedures. Furthermore, it is possible to automatically translate from Scheme procedures to hygienic, composable macros. Syntax-rules become object, assembler code.

The approach of this talk makes programming of Scheme macros and of other related head-first re-writing systems more like a craft than a witchcraft.