How to Write Seemingly Unhygienic and Referentially Opaque Macros with Syntax-rules

- Overview
- Petrofsky Extraction
  extracting variables from arguments of a macro
  - weakly unhygienic syntax-rules
- Macro, redefine thyself!
- Infected λ
- Discussion
  - Is it legal?
  - What is hygiene, really?
  - More macros are possible

This talk will make several assertions that might be hard to believe. I will say the most incredible statement right upfront however. This whole subject of writing seemingly referentially opaque macros with syntax rules has been *actually* inspired by a practical application. Yes, indeed. I'd like to talk about the history for a moment, because it gives the overview of the paper.

How it all began

Antti Karttunen "Dirty macros with define-syntax?"
comp.lang.scheme, Mar 6, 2002.

```
(define-convform (NUKE_CRS (I ifile) (O ofile))
  (run (tr -d "\\015") (< ,ifile) (> ,ofile)))
```

```
(let ((IO-ENV '((file1 "/tmp/a"))))
  (NUKE_CRS (I file1) (O file2))
  (PRINT (I file2)))
```

Can the IO environment for file labels be lexical?
Can you do it without global variables?

It all started with an article Antti Karttunen posted on the newsgroup comp.lang.scheme on Mar 6, 2002. {The title of the article was "Dirty macros with define-syntax?".} The subject is a domain-specific language for assembling "pipelines" of system commands in SCSH. Unlike SCSH own pipelines, the new pipelines could allow more than one input or output, allow single-stepping for debugging, and should use temporary files to avoid losing the work should a step fail. In particular, to avoid passing many parameters to a step macro, Antti wanted to refer to the parameters, IO-ENV, implicitly. He needed a macro whose expansion could refer to lexically bound variables [here]. Because SCSH (Scheme48, actually) encourages R5RS macros, Antti was asking how to implement his constructions with syntax-rules. Several people, including Al Petrofsky, posted replies saying that R5RS macros are referentially transparent, so this cannot be done. I posted an implementation of define-convform and side-stepped this issue by using the global IO-ENV. On March 20, Antti replied to me saying that it was all nice but he wanted to avoid too many globals in his code. Specifically, he wanted {this: (let ((IO-ENV...)))}.

Hygienic macros cannot expand in referentially-opaque code: if a macro expansion introduces a free identifier, the identifier refers to the binding occurrence which was in effect when the macro was defined rather than when the macro was expanded. E.g.

```
(define foo 1)
(define-syntax mfoo
  (syntax-rules ()
    ((mfoo) foo)))

(let ((foo 2)) (mfoo)) ;==> 1
```

If `foo` was not defined when macro `mfoo` was introduced, `(mfoo)` will always lead to an "unbound identifier" error.

Still, there is a way to get around.

[hide the last line]

On Friday morning Mar 22 I started to type my reply that I agree with Al Petrofsky: you can't write a R5RS macro that allows an inserted identifier to be captured by a local binding. I gave an example: {Hygienic macros cannot expand in referentially-opaque code:} if a macro expansion introduces a free identifier [here], the identifier refers to the binding occurrence which was in effect when the macro was defined [here] rather than when the macro was expanded [as in here].

You can't defy gravity. As I was typing that, it struck me: And why not? What if I use my own binding form, `mylet` rather than the regular `let`? That mylet will expand into the regular let, *and* redefine the macro in question right after that, at this spot. **[show on the figure, how macro mylet would expand into a let with the definition of mfoo within its scope]** The identifier foo in the macro-expansion would appear to be captured by the local, mylet binding?

That silly idea made me spend several hours trying it out. Eventually, I got such mylet macro to work. Then I realized it had a flaw. You cannot nest the mylet bindings. It was around 3pm, I decided I wasted enough time and should cut losses. I had a lunch, read something, and started to wrap up my message to say that what Antti wants cannot be done. I had a trick, but it doesn't work. As I was typing that I remembered Al Petrofsky posted an article half a year earlier with a syntax-rule implementation of a loop-until-exit form. Al mentioned he also had a problem with nesting, and he solved it. I wondered if his solution can work for mylet too. So I checked Al's article from Google, read it, started to implement what I've understood. By 11 pm, it seemed to work out, So I finished my message to Antti by typing:

[reveal the phrase] That phrase was followed by a couple of pages of dense code with denser comments, and a few examples,

```
(mylet ((foo 3)) (mylet ((foo 4))
   (mylet ((foo 5)) (list foo (mfoo)))))
; ==> (5 5)
```

`(mfoo)` captures the binding of `foo` from the *lexically* closest environment.

```
(mylet ((foo 3))
 (let ((thunk (lambda () (mfoo))))
  (mylet ((foo 4)) (list foo (mfoo) (thunk)))))
;==> (4 4 3)
```

BTW, it's even possible to redefine let and replace it with our mylet, so you can write

```
(let ((foo 2)) (let ((foo 3))
    (let ((foo 4)) (list (mfoo) foo))))
```

and get the same apparently non-hygienic results. *It's just the matter of defining our macros at a high-enough level...*

As you see, `(mfoo)` captures the binding of `foo` from the *lexically* closest environment: here and here and here. I used Bigloo to run all the examples.

BTW, it's even possible to redefine `let` and replace it with our `mylet`, so you can write (let ...) and get the same apparently non-hygienic results. It's just the matter of defining our macros at a high-enough level...

As I was falling asleep later that night I realized that I needed to subvert only the lambda form. The let forms will get in line.

On Sunday Antti replied asking if I mind if he brings the discussion back to public. I guess you see now who to blame for this.

How to Write Seemingly Unhygienic and Referentially Opaque Macros with Syntax-rules

- Overview
- Petrofsky Extraction
  extracting variables from arguments of a macro
  - weakly unhygienic syntax-rules
- Macro, redefine thyself!
- Infected λ
- Discussion
  - Is it legal?
  - What is hygiene, really?
  - More macros are possible

Well, this basically was all the talk. In the remaining time, I will repeat it again with more details. We will thoroughly discuss Petrofsky extraction, because it can be used to write weakly unhygienic syntax-rules, because we will use it to write seemingly referentially opaque macros, and because it's so cool. We will show that a macro that introduces a binding and then redefines itself and other macros leads to the overall referential opaqueness. We will get a lambda to do such redefinitions, and then make the let binding forms to use the subverted lambda.

The end result demonstrates a syntax-rule macro that looks exactly like a careless, referentially opaque Lisp-style macro. The final section discusses what it all means: for macro writers, for macro users, and for programming language researchers.

But first a word about terminology.

Hygiene

Avoiding inadvertent captures of free variables through systematic renaming

Narrow sense: HC/ME
"Generated identifiers that become binding instances in the completely expanded program must only bind variables that are generated at the same transcription step" [KFFD86]

```
(define-syntax mbi
  (syntax-rules ()
    ((_ body) (let ((i 10)) body))))

(let ((i 1)) (mbi (* 1 i)))
; =/=> (let ((i 1)) (let ((i 10)) (* 1 i)))
; ==>  (let ((i~2 1))
         (let ((i~5 10)) (* 1 i~2)))
```

A macro system is called hygienic, in the general sense, if it avoids inadvertent captures of free variables through systematic renaming. The free variables in question can be either generated variables, or variables present in macro invocations (i.e., user variables). Here, this `i` is a user variable, it came from an argument of a macro. This `i` was a free variable in the macro transformer itself. It's called a generated identifier. A narrowly defined hygiene is avoiding the capture of user variables by generated bindings. The precise definition, a hygiene condition for macro expansions (HC/ME), is given in the paper whose two authors are in the audience. {We borrowed all the terminology from that paper, which is most precise. HC/ME says: 'Generated identifiers that become binding instances in the completely expanded program must only bind variables that are generated at the same transcription step.'}

If we naively expand `(mbi (* 1 i))` – match the expression with the pattern, substitute the matched body in the template – we will get this. This `i` is a generated identifier, which is also a generated binding. This `i` is a non-generated identifier reference. Here it is captured by the generated binding. But HC/ME prohibits this. Therefore, expansion occurs as **[that]** and gives the result 1. The identifier `i~2` is different from `i~5`: we will call them identifiers of different *colors*.

We have talked about generated bindings capturing user identifiers. The opposite case is user binding capturing generated identifiers, which is called referential opaqueness. We saw the example of that earlier.

Petrofsky extraction: Breaking the weak hygiene

Goal:

```
(mbi 10 (* 1 i)) ==> (let ((i 10)) (* 1 i))
```

Does not work:

```
  (define-syntax mbi
    (syntax-rules ()
      ((_ val body) (let ((i val)) body))))
  (mbi 10 (* 1 i)) ;==> (let ((i~5 10)) (* 1 i))
```

Does work:

```
  (define-syntax mbi-i
    (syntax-rules ()
      ((_ i val body) (let ((i val)) body))))
  (mbi-i i 10 (* 1 i)) ;==> (let ((i 10)) (* 1 i))
```

As we said earlier, Petrofsky extraction is what made our trick possible. It is a very useful technique and deserves careful explanation. I will now describe how I understand and implement it. I urge you to read the original articles by Al Petrofsky.

Our goal here is to break a weak hygiene. We want to write such a macro `mbi` so that this expression expands into that expression, with let-binding capturing the variable in the macro's argument. We assume that `i` is not defined at the global level – or defined before macro mbi and not redefined since. This assumption is what distinguishes a weak hygiene.

We already saw that this definition for `mbi` does not work for us. The macro expands to this, and no capture occurs, just as HC/ME says. However, if we explicitly pass the variable to capture, it works. Indeed, here a non-generated binding caught a non-generated reference – which does not violate hygiene. So, we should think how to convert this (the first) invocation into that (the second) one. In here (second expression), this `i` and that `i` are the same. So, to make the conversion, we need to extract `i` from the argument of `mbi`, and we're done.

Petrofsky extraction: `extract` and `extract*`

```
extract SYMB BODY (K-HEAD K-IDL . K-ARGS)
==> (K-HEAD (extr-id .  K-IDL) . K-ARGS)
```

```
  (define-syntax extract (syntax-rules ()
   ((_ symb body _cont)
    (letrec-syntax
     ((tr
       (syntax-rules (symb)
        ((_ x symb tail
              (cont-head symb-l . cont-args))
         (cont-head (x . symb-l) . cont-args))
        ((_ d (x . y) tail cont)
         (tr x x (y . tail) cont))
        ((_ d1 d2 ()
            (cont-head  symb-l . cont-args))
         (cont-head (symb . symb-l) . cont-args))
        ((_ d1 d2 (x . y) cont)
         (tr x x y cont)))))
    (tr body body () _cont)))))
```

Thus we come to the macro extract, which extracts a colored identifier from a form BODY. To be more precise, we extract from the BODY an identifier that refers to the same binding occurrence as SYMB. SYMB and the extracted identifier may be of different colors, but refer to the same binding occurrence. William Clinger's paper 'Macros that work' explains how that may happen.

The third argument is a continuation of that shape. The macro `extract` expands into the following, where this is an extracted id.

Let us look into the macro. All the work is done by a helper tr. Its first two arguments are always the same. We use the second argument to match with the symbol. If it matches {1st clause}, we take the first argument and we're done. We are satisfied with the first occurrence of the desired id.

if this {1st clause} didn't match, we see if the form is a pair. If so, we put the tail into the worklist and go check the head. If the form is not a pair and the worklist is empty {the third clause}, well, we found nothing and we just return the original symbol as an 'extracted' id. If the worklist is not empty {4th clause}, we deal with it.

We will often be using `extract*`, which is similar but extracts several identifiers. It takes a list of identifiers here and expands into the same kind of form, with the corresponding list of extracted identifiers here. All the work is done by this macro, `extract`. `extract*` merely does map in CPS.

Petrofsky extraction: mbi-dirty-v1 – the first weakly unhygienic macro

```
(define-syntax mbi-dirty-v1
  (syntax-rules ()
    ((_ _val _body)
      (let-syntax
        ((cont
          (syntax-rules ()
            ((_ (symb) val body)
              (let ((symb val)) body) ))))
        (extract i _body
                  (cont () _val _body))))))

(mbi-dirty-v1 10 (* 1 i))
;==> (let ((i~11 10)) (* 1 i~11))
```

This macro carries out our program of converting `mbi` into `mbi-i`. We extract `i` from the body and bind it in the let form. Indeed, it seems to work. The colors in the expansion are right, the capturing occurs, and the evaluation result is 10.

Petrofsky extraction: the flaw of mbi-dirty-v1

The macro `mbi-dirty-v1` does not nest

```
(mbi-dirty-v1 10
  (mbi-dirty-v1 20 (* 1 i)))

;==>

(let ((i~16 10))
  (let ((i~17~25~28 20)) (* 1 i~16)))
```

But the macro has a flaw: it does not nest. This expands into that and gives 10 rather than to 20 as we might have hoped. The outer invocation of `mbi-dirty-v1` creates a binding for `i` – which violates the weak hygiene assumption. Petrofsky has shown how to overcome this problem as well: we need to re-define `mbi-dirty-v1` in the scope of the new binding to `i`. Hence we need a macro that re-defines itself in its own expansion. We however face a problem: If the outer invocation of `mbi-dirty-v1` re-defines itself, this redefinition has to capture the inner invocation of `mbi-dirty-v1`. We already know how to do that, by extracting the colored identifier `mbi-dirty-v1` from the outer macro's body. We need thus to extract two identifiers: `i` and `mbi-dirty-v1`. We arrive at the following code:

Petrofsky extraction: mbi-dirty-v2 – a weakly unhygienic macro

```
(define-syntax mbi-dirty-v2 (syntax-rules ()
  ((_ _val _body)
   (letrec-syntax
     ((doit (syntax-rules ()
       ((_ (myself-symb i-symb) val body)
        (let ((i-symb val))
         (let-syntax
          ((myself-symb
            (syntax-rules ()
             ((_ val__ body__)
              (extract*
                (myself-symb i-symb) body__
               (doit () val__ body__))))))
          body))))))

     (extract* (mbi-dirty-v2 i) _body
               (doit () _val _body))))))
```

As we have said: we start by extracting our own id from the body {see extract*} make the binding {see let}, redefine oneself {let-syntax} and leave the body. There is something here resembling a bootstrapping. `doit` is the continuation from extract*

Petrofsky extraction: mbi-dirty-v2 – a weakly unhygienic macro

```
(mbi-dirty-v2 10
  (mbi-dirty-v2 20 (* 1 i)))
;===> 20
```

But:

```
(let ((i 1))
  (mbi-dirty-v2 10 (* 1 i)))
; ==> (let ((i~26 10))
        (let ((i~52 20)) (* 1 i~52)))
; ===> 1
```

Now, the nesting works. However, the macro `mbi-dirty-v2` is still only weakly unhygienic. If we evaluate (let ((i 1)) ...) **[show]** we obtain **[show]** which evaluates to 1 rather than 10.

We will now turn to referential transparency – the topic of this talk.

Apparent referential opacity with mylet

```
(mylet ((foo 2))
  (mylet ((foo 3)) (list foo (mfoo))))

;==>
(let ((foo 2))
  (define-syntax-mfoo-to-expand-into-foo)
  (re-define-mylet-to-account-for-
      redefined-foo-and-mfoo)
  (let ((foo 3))
    (define-syntax-mfoo-to-expand-into-foo)
    (re-define-mylet-to-account-for-
        redefined-foo-and-mfoo)
    (list foo (mfoo))
    ))

 ;===> (3 3)
```

Earlier we showed a custom binding form `mylet` and what it can do. `(mfoo)` is a macro that expands in the identifier `foo`. We want this id be captured by the closest lexical binding, like this one.

mylet is a custom binding form that helps us. We want this whole expression to expand as follows: make a binding, redefine `mfoo` so that it would refer to this `foo`, redefine oneself. Again, make a binding, redefine `mfoo`, redefine oneself. Therefore, this `mfoo` will generate the same `foo` reference as this one. The end result will be this {(3 3)}. This process of defining and redefining looks awfully close to what we have just seen in the case of the weakly unhygienic macro. So, we can use it, with a couple of adjustments, as shown in the paper.

Can we do better? Rather than introducing the custom binding `mylet`, can we overload regular `let` forms to do the same? Yes.

Implementation of mylet

```
(define-syntax mylet (syntax-rules ()
  ((_ ((_var _init)) _body)
   (letrec-syntax
     ((doit (syntax-rules ()
        ((_ (mylet-symb mfoo-symb foo-symb)
            ((var init)) body)
          (let ((var init))
            (make-mfoo mfoo-symb foo-symb
              (letrec-syntax
                ((mylet-symb
                  (syntax-rules ()
                    ((_ ((var_ init_)) body_)
                     (extract*
                      (mylet-symb mfoo-symb
                       foo-symb) (var_ body_)
                      (doit () ((var_ init_))
                            body_))))))
                body)))))))
  (extract* (mylet mfoo foo) (_var _body)
      (doit () ((_var _init)) _body)))))
```

[backup slide]

The macro follows our plan: doing `let`, redefining `mfoo`, redefining ourselves. We have to be sure we do it with rightly colored ids. That's what the rest of the code is for. `doit` is the continuation from `extract*`. All the examples we showed in the beginning run.

Can we do better? Can we redefine all binding forms? Yes.

Macro `defile`

```
(define-syntax defile
  (syntax-rules ()
    ((_ dbody)
     (letrec-syntax
         ((do-defile
           (syntax-rules ()
             ; all the overloaded symbols
             ((_ (let-symb let*-symb
                  letrec-symb lambda-symb
                  mfoo-symb foo-symb)
                 body-to-defile)
          ; ... 62 more long lines ...
       (extract* (let let* letrec lambda
                  mfoo foo) dbody
                 (do-defile () dbody))
       ))))
```

This is the job of a macro defile, which is schematically shown here. As we see, we extract and re-define all binding forms. The definitions for our let, letrec and let* are exactly like the ones in R5RS – only they use our lambda. Our lambda makes bindings with the help of the original lambda, and then redefines all the binding forms and the targeted macro `mfoo` in the scope of the new binding. It really looks as if lambda were infected by a virus. Every occurrence of lambda transcribes the corrupted gene and proliferates the virus further and further. Too bad that the defile macro is long and the time is short. I'll show only one example of using the defile macro.

A defiled example

```
(defile
  (let* ((foo 2)
         (i 3)
         (foo 4)
         ; will capture binding of foo to 4
         (ft (lambda () (mfoo)))
         (foo 5)
         ; will capture the arg of ft1
         (ft1 (lambda (foo) (mfoo)))
         (foo 6))
    (list foo (mfoo) (ft) (ft1 7) '(mfoo))))
; ==> (6 6 4 7 (mfoo))
```

As if

```
(define-macro (mfoo) foo)
```

The foo in the expansion of mfoo is indeed captured by the closest lexical binding: here it is 6, here it is 4, and here it is 7. All in all, mfoo behaves as if it, unless quoted, were just the identifier foo. In other words, as if mfoo were defined as a non-hygienic, referentially opaque macro.

To be able to capture a generated identifier by a local binding, we need to know the name of that identifier and the name of a macro that generates it. Macro defile has these identifiers hard-coded as `foo` and `mfoo`. Nothing of course prevents us from accepting these names from the user. We thus arrive at

let-leaky-syntax – *library* syntax

```
(display
 (let-leaky-syntax
  quux
  ((mquux (syntax-rules ()
     ((_ val) (+ quux quux val)))))

  (let* ((bar 1) (quux 0) (quux 2)
         (lquux (lambda (x) (mquux x)))
         (quux 3)
         (lcquux (lambda (quux) (mquux quux))))
     (list (+ quux quux) (mquux 0) (lquux 2)
           (lcquux 5)))))

;==> (6 6 6 15)
```

So much work in being able to say this one word: library.

The form let-leaky-syntax is similar to let-syntax. But let-leaky-syntax takes an additional first argument, an identifier from the body of the defined syntax-rules to be captured by the closest lexical binding. As the examples show, the variable is captured indeed. In particular, the macro `mquux` expands to an expression that adds the value of an identifier `quux` twice to the value of the mquux's argument. Because the identifier quux is declared special, that is, to be captured by the closest local binding, a procedure `(lambda (quux) (mquux quux))` effectively triples its argument. One quux comes from here, and two others – again from here {the argument}. And triple it does.

Now, discussion.

Is it legal?

- Redefinitions of `let` and `lambda` are not prohibited
- Only R5RS facilities are used
- Undefined behavior is not relied upon
- `let`, `letrec`, `let*`, and `lambda` relate precisely as those in R5RS

What about the R5RS hygiene condition?

"If a macro transformer inserts a free reference to an identifier, the reference refers to the binding that was visible where the transformer was specified, regardless of any local bindings that may surround the use of the macro." [R5RS]

```
(define foo 1)
(defile
  (let ((foo 2)) (list (mfoo) foo)))
```

One can argue that our re-defined lambda leads to a violation of the constraint R5RS places on the macro system: ”If a macro transformer inserts a free reference to an identifier, the reference refers to the binding that was visible where the transformer was specified, regardless of any local bindings that may surround the use of the macro.” This paragraph however applies exactly as it is to the defiled macros. In this code, the identifier `foo` inserted by the expansion of the macro `mfoo` indeed refers to the binding of `foo` that was visible when the macro `mfoo` was defined. The twist is that the definition of the macro `mfoo` happened right after the local binding of foo. Despite `mfoo` being an R5RS, referentially transparent macro, the overall result *looks* like the expansion of a referentially opaque macro.

Isn't it just a Scheme-*like* little language?

- Every macro is a syntactic *extension*
- Such a little language was presumed **impossible** with syntax-rules

The macro defile and let-leaky-syntax indeed have to surround the victim's code. One can therefore object if we merely create our own 'little language' that resembles Scheme but does not guarantee the referential transparency of macro expansions. Any macro by definition extends the language. The extended language is still expected to obey certain constraints. The impetus for hygienic macros was specifically to create a macro system with guaranteed hygienic constraints. Although syntax-rules are Turing complete, certain computations, for example, determining if two identifiers are spelled the same, are outside of their scope. It was a common belief therefore that syntax-rules are *thoroughly* hygienic. In particular, the common interpretation of hygiene conditions from the 'Macros that work' paper precludes let-leaky-syntax and similar extensions.

What is a hygiene then?

A good question

We conclude that the subject of macro hygiene is not at all decided, and more research is needed to precisely state what hygiene formally means and which precisely assurances it provides.

Practical conclusions

- let-leaky-syntax library form:
  A new class of macros without resorting to lower-level facilities

- Encouragement

- Principle: can't change a person $\Longrightarrow$ change his environment

For a practical programmer, we offer a let-leaky-syntax library form. The programmer can therefore define a class of powerful syntactic extensions with standard R5RS syntax-rules, without resorting to lower-level macro facilities. In general, the practical macro programmer will hopefully view the conclusions of this paper as an encouragement. We should realize the informal and narrow nature of many assertions about R5RS macros. We should not read into R5RS more than it actually says. Thus we can write more and more expressive syntax-rules than we were previously led to believe.

macro-expand-time environments

```
(make-env
 (bind ((a 1))
   (bind ((b 2))
     (list (lookup a) (lookup b)
       (bind ((a 3))
         (list (lookup a) (lookup b)))))))

; ==> (list 1 2 (list 3 2))
```

Macro make-env is to maintain macro-expand-time environment. You can bind values in the environment (associate them with keys) – and then look them up by the key. You can shadow one binding with another. For example, here. The example looks and feels as if `bind` were just `let`. However, here the binding and the lookup occurs at macroexpand time. This whole code expands to this.

Complex identifiers

```
(make-env1
 (lambda ((id a b c))
   (lambda ((id a b))
     (list (id a b c) (id a b)
           '(id a b x) '(id a b) 'x))))

; =equiv=>
(lambda (temp~1)
  (lambda (temp~2)
    (list temp~1 temp~2
       'abx 'ab 'x)))
```

A program is a closed term. The meaning of a program does not change if all identifiers are alpha-renamed. Therefore, identifier doesn't have to be a symbol. It can be anything that can be compared in equality. Al Petrofsky has pointed out this approach to me – which has arisen in our previous conversation and was implemented in my universally portable case-sensitive symbols.

A macro make-env1 extends this idea for identifiers-as-lists. The insight the key in the syntactic, macro-expand-time environment don't have to be symbols. make-env1 uses lists of symbols as keys, and identifiers as values.

A form `(lambda ((id a b ...))  body)` binds a key `(a b c ...)` to a value `temp~xxx`. The value is a temporary, colored identifier generated by syntax-rules macros. The hygiene property makes such identifiers unique. The macro make-env1 re-writes `(lambda ((id a b ...)) body)` into `(lambda (temp~xxx) re-written-body)`. Hence we can indeed use composite identifiers in the body of lambda: `(id a b ...)`. Here `id` is a synonym for 'lookup'. The expansion of `id` in the *current environment* (maintained implicitly by the macro make-env1) replaces `(id a b ...)` with the corresponding value – which is the identifier `temp~xxx` – the same identifier that was bound by lambda. That was the critical condition that makes everything work.

Incidentally, the representation of identifiers by complex structures isn't outrageous. Kohlbecker's algorithm does a similar thing: The algorithm represents identifiers as vectors. In our case, an identifier is an S-expression that is distinguished by a head symbol `id`. The rest is a list of one-character symbols that spell the identifier name. See, the quoted symbols work as expected.

Really dirty stuff

A syntax-rule that *generates* ids

```
((((make-env1
    (let-syntax
        ((add-c
          (syntax-rules ()
            ((_ id a b) (id a b c)))))
      (lambda ((id a b c))
        (lambda ((id a b))
          (lambda ((id c))
            (list (id a b) (id c)
                  (add-c id a b)))))
      ))
1) 2) 3)
;==> (list 2 3 1)
```