Macro Debugger: Inspecting Macro Expansion

Version 9.4.0.3

Ryan Culpepper

October 27, 2026

The macro-debugger collection contains two tools: a stepper for macro expansion and a standalone syntax browser. The macro stepper shows the programmer the expansion of a program as a sequence of rewriting steps, using the syntax browser to display the individual terms. The syntax browser uses colors and a properties panel to show the term's syntax properties, such as lexical binding information and source location.

1 Macro Stepper

```
(require macro-debugger/stepper)
```
package: macro-debugger

```
(expand/step stx) → void?
  stx : any/c
```

Expands the syntax (or S-expression) and opens a macro stepper frame for stepping through the expansion.

```
(expand-module/step mod) → void?
  mod : module-path?
```

Expands the source file named by *mod*, which must contains a single module declaration, and opens a macro stepper frame for stepping through the expansion.

```
(macro-stepper-repl [new-repl?
                     #:eval? eval?]) → void?
  new-repl? : any/c = #f
  eval? : any/c = #t
```

Creates a macro stepper frame and starts a read-eval-print loop that shows the expansion of every expression entered into the repl. If *new-repl?* is true, a new repl is created by calling `(read-eval-print-loop)`; otherwise, the current repl is reused. If *eval?* is true, then expressions are evaluated after expansion; otherwise only the compile-time parts are evaluated.

The repl is implemented by installing a custom evaluation handler that chains to the original handler to do evaluation.

2 Macro Expansion Tools

```
(require macro-debugger/expand)
    package: macro-debugger-text-lib
```

This module provides `expand`-like procedures that allow the user to specify macros whose expansions should be hidden.

Warning: because of limitations in the way macro expansion is selectively hidden, the resulting syntax may not evaluate to the same result as the original syntax.

```
(expand-only stx transparent-macros) → syntax?
  stx : any/c
  transparent-macros : (listof identifier?)
```

Expands the given syntax *stx*, but only shows the expansion of macros whose names occur in *transparent-macros*.

Example:

```
> (syntax->datum
   (expand-only #'(let ([x 1] [y 2]) (or (even? x) (even? y)))
                (list #'or)))
'(let ((x 1) (y 2))
   (let ((or-part (even? x))) (if or-part or-part (#%expression
(even? y)))))
```

```
(expand/hide stx hidden-macros) → syntax?
  stx : any/c
  hidden-macros : (listof identifier?)
```

Expands the given syntax *stx*, but hides the expansion of macros in the given identifier list (conceptually, the complement of expand-only).

Example:

```
> (syntax->datum
   (expand/hide #'(let ([x 1] [y 2]) (or (even? x) (even? y)))
                (list #'or)))
'(let-values (((x) '1) ((y) '2)) (or (#%app even? x) (#%app even?
y)))
```

```
(expand/show-predicate stx show?) → syntax?
  stx : any/c
  show? : (-> identifier? boolean?)
```

Expands the given syntax *stx*, but only shows the expansion of macros whose names satisfy the predicate *show?*.

Example:

```
> (syntax->datum
   (expand/show-predicate
    #'(let ([x 1] [y 2]) (or (even? x) (even? y)))
    (lambda (id) (memq (syntax-e id) '(or #%app)))))
'(let ((x 1) (y 2))
   (let ((or-part (#%app even? x)))
     (if or-part or-part (#%expression (#%app even? y)))))
```

3 Macro Stepper API for Macros

```
(require macro-debugger/emit)
```
package: macro-debugger

Macros can explicitly send information to a listening macro stepper by using the procedures in this module.

```
(emit-remark fragment ... [#:unmark? unmark?]) → void?
  fragment : (letrec ([emit-arg/c
                       (recursive-contract
                        (or/c string?
                              syntax?
                              (listof emit-arg/c)
                              (-> emit-arg/c)))])
               emit-arg/c)
  unmark? : boolean? = (syntax-transforming?)
```

Emits an event to the macro stepper (if one is listening) containing the given strings and syntax objects. The macro stepper displays a remark by printing the strings and syntax objects above a rendering of the macro's context. The remark is only displayed if the macro that emits it is considered transparent by the hiding policy.

By default, syntax objects in remarks have the transformer's mark applied (using `syntax-local-introduce`) so that their appearance in the macro stepper matches their appearance after the transformer returns. Unmarking is suppressed if *unmark?* is `#f`.

```
(define-syntax (mymac stx)
  (syntax-case stx ()
    [(_ x y)
     (emit-remark "I got some arguments!"
                  #'x
                  "and"
                  #'y)
     #'(list 'x 'y)]))
(mymac 37 (+ 1 2))
```

(Run the fragment above in the macro stepper.)

```
(emit-local-step before after #:id id) → void?
  before : syntax?
  after : syntax?
  id : identifier?
```

Emits an event that simulates a local expansion step from *before* to *after*.

The *before* and *after* terms are marked with `syntax-local-introduce` so they appear in the macro stepper like they would if the step were truly generated from a local expansion.

The *id* argument acts as the step's "macro" for the purposes of macro hiding.

4 Macro Stepper Text Interface

```
(require macro-debugger/stepper-text)
         package: macro-debugger-text-lib
```

```
(expand/step-text stx [show?]) → void?
  stx : any/c
  show? : (or/c (-> identifier? boolean?) = (lambda (x) #t)
                (listof identifier?))
```

Expands the syntax and prints the macro expansion steps. If the identifier predicate is given, it determines which macros are shown (if absent, all macros are shown). A list of identifiers is also accepted.

Example:

```
> (expand/step-text #'(let ([x 1] [y 2]) (or (even? x) (even? y)))
                    (list #'or))
Macro transformation
(let ((x 1) (y 2)) (or (even? x) (even? y)))
  ==>
(let
 ((x 1) (y 2))
 (let:1 ((or-part:1 (even? x))) (if:1 or-part:1 or-part:1 (or:1
(even? y)))))

Macro transformation
(let
 ((x 1) (y 2))
 (let:1 ((or-part:1 (even? x))) (if:1 or-part:1 or-part:1 (or:1
(even? y)))))
  ==>
(let
 ((x 1) (y 2))
 (let:1
  ((or-part:1 (even? x)))
  (if:1 or-part:1 or-part:1 (#%expression:2 (even? y)))))
```

```
(stepper-text stx [show?]) → (symbol? -> void?)
  stx : any/c
  show? : (or/c (-> identifier? boolean?) = (lambda (x) #t)
                (listof identifier?))
```

Returns a procedure that can be called on the symbol `'next` to print the next step or on the symbol `'all` to print out all remaining steps.

5 Syntax Browser

```
(require macro-debugger/syntax-browser)
```
package: macro-debugger

```
(browse-syntax stx) → void?
  stx : syntax?
```

Creates a frame with the given syntax object shown. More information on using the GUI is available below.

```
(browse-syntaxes stxs) → void?
  stxs : (listof syntax?)
```

Like browse-syntax, but shows multiple syntax objects in the same frame. The coloring partitions are shared between the two, showing the relationships between subterms in different syntax objects.

6 Using the Macro Stepper

6.1 Navigation

The stepper presents expansion as a linear sequence of rewriting process, and it gives the user controls to step forward or backwards as well as to jump to the beginning or end of the expansion process.

If the macro stepper is showing multiple expansions, then it also provides “Previous term” and “Next term” buttons to go up and down in the list of expansions. Horizontal lines delimit the current expansion from the others.

6.2 Macro Hiding

Macro hiding lets one see how expansion would look if certain macros were actually primitive syntactic forms. The macro stepper skips over the expansion of the macros you designate as opaque, but it still shows the expansion of their subterms.

The bottom panel of the macro stepper controls the macro hiding policy. The user changes the policy by selecting an identifier in the syntax browser pane and then clicking one of “Hide module”, “Hide macro”, or “Show macro”. The new rule appears in the policy display, and the user may later remove it using the "Delete" button.

The stepper also offers coarser-grained options that can hide collections of modules at once. These options have lower precedence than the rules above.

Macro hiding, even with no macros marked opaque, also hides certain other kinds of steps: internal defines are not rewritten to letrecs, begin forms are not spliced into module or block bodies, etc.

7 Using the Syntax Browser

7.1 Selection

The selection is indicated by bold text.

The user can click on any part of a subterm to select it. To select a parenthesized subterm, click on either of the parentheses. The selected syntax is bolded. Since one syntax object may occur inside of multiple other syntax objects, clicking on one occurrence will cause all occurrences to be bolded.

The syntax browser displays information about the selected syntax object in the properties panel on the right, when that panel is shown. The selected syntax also determines the highlighting done by the secondary partitioning (see below).

7.2 Primary Partition

The primary partition is indicated by foreground color.

The primary partitioning always assigns two syntax subterms the same color if they have the same marks. In the absence of unhygienic macros, this means that subterms with the same foreground color were either present in the original pre-expansion syntax or generated by the same macro transformation step.

Syntax colored in black always corresponds to unmarked syntax. Such syntax may be original, or it may be produced by the expansion of a nonhygienic macro.

Note: even terms that have the same marks might not be `bound-identifier=?` to each other, because they might occur in different environments.

7.3 Secondary Partitioning

The user may select a secondary partitioning through the Syntax menu. This partitioning applies only to identifiers. When the user selects an identifier, all terms in the same equivalence class as the selected term are highlighted in yellow.

The available secondary partitionings are:

- `bound-identifier=?`
- `free-identifier=?`

7.4 Properties

When the properties pane is shown, it displays properties of the selected syntax object. The properties pane has two tabbed pages:

- **Term**:

  If the selection is an identifier, shows the binding information associated with the syntax object. For more information, see `identifier-binding`, etc.

- **Syntax Object**:

  Displays source location information and other properties (see `syntax-property`) carried by the syntax object.

7.5 Interpreting Syntax

The binding information of a syntax object may not be the same as the binding structure of the program it represents. The binding structure of a program is only determined after macro expansion is complete.

8 Finding Useless `requires`

```
(require macro-debugger/analysis/check-requires)
```
package: `macro-debugger-text-lib`

The "Check Requires" utility can be run as a raco subcommand. For example (from racket root directory):

```
raco check-requires racket/collects/syntax/*.rkt
```

```
raco check-requires -kbu openssl
```

Each argument is interpreted as a file path if it exists; otherwise, it is interpreted as a module path. See `check-requires` for a description of the output format, known limitations in the script's recommendations, etc.

```
(check-requires  module-to-analyze
                [#:show-keep? show-keep?
                 #:show-bypass? show-bypass?
                 #:show-drop? show-drop?
                 #:show-uses? show-uses?])    → void?
  module-to-analyze : module-path?
  show-keep? : boolean? = #t
  show-bypass? : boolean? = #t
  show-drop? : boolean? = #t
  show-uses? : boolean? = #f
```

Analyzes *module-to-analyze*, detecting useless requires. Each module imported by *module-to-analyze* is classified as one of KEEP, BYPASS, or DROP. For each required module, one or more lines is printed with the module's classification and supporting information. Output may be suppressed based on classification via *show-keep?*, *show-bypass?*, and *show-drop?*.

When run via `raco check-requires`, only DROP recommendations are printed by default.

Modules required `for-label` are not analyzed.

```
KEEP req-module at req-phase
```

The require of module *req-module* at phase *req-phase* must be kept because bindings defined within it are used.

If *show-uses?* is true, the dependencies of *module-to-analyze* on *req-module* are enumerated, one per line, in the following format:

```
exp-name at use-phase (mode ...) [RENAMED TO ref-name]
```

Indicates an export named *exp-name* is used at phase *use-phase* (not necessarily the phase it was provided at, if *req-phase* is non-zero).

The *modes* indicate what kind(s) of dependencies were observed: used as a `reference`, appeared in a syntax template (`quote-syntax`), etc.

If the `RENAMED TO` clause is present, it indicates that the binding is renamed on import into the module, and *ref-name* gives the local name used (*exp-name* is the name under which *req-module* provides the binding).

`BYPASS` *req-module* `at` *req-phase*

The require is used, but only for bindings that could be more directly obtained via one or more other modules. For example, a use of `racket` might be bypassed in favor of `racket/base`, `racket/match`, and `racket/contract`, etc.

A list of replacement requires is given, one per line, in the following format:

`TO` *repl-module* `at` *repl-phase* `[WITH RENAMING]`

Add a require of *repl-module* at phase *repl-phase*. If *show-uses?* is true, then following each `TO` line is an enumeration of the dependencies that would be satisfied by *repl-module* in the same format as described under `KEEP` below.

If the `WITH RENAMING` clause is present, it indicates that at least one of the replacement modules provides a binding under a different name from the one used locally in the module. Either the references should be changed or `rename-in` should be used with the replacement modules as necessary.

Bypass recommendations are restricted by the following rules:

- *repl-module* must not involve crossing a new `private` directory from *req-module*
- *repl-module* is never a built-in ("`#%`") module
- *req-module* must not be in the "no-bypass" whitelist

`DROP` *req-module* `at` *req-phase*

The require appears to be unused, and it can probably be dropped entirely.

Due to limitations in its implementation strategy, `check-requires` occasionally suggests dropping or bypassing a module that should not be dropped or bypassed. The following are typical reasons for such bad suggestions:

- The module's invocation has side-effects. For example, the module body may update a shared table or perform I/O, or it might transitively require a module that does. (Consider adding the module to the whitelist.)

- Bindings from the module are used in identifier comparisons by a macro, such as appearing in the macro's "literals list." In such cases, a macro should annotate its expansion with the `'disappeared-use` property containing the identifier(s) compared with its literals; however, most casually-written macros do not do so. On the other hand, macros and their literal identifiers are typically provided by the same module, so this problem is somewhat uncommon.

Examples:

```
> (check-requires 'framework)
KEEP racket/base at 0
KEEP racket/contract/base at 0
KEEP racket/unit at 0
KEEP racket/class at 0
KEEP racket/gui/base at 0
KEEP racket/set at 0
KEEP mred/mred-unit at 0
KEEP framework/framework-unit at 0
KEEP framework/private/sig at 0
KEEP scribble/srcdoc at 0
KEEP framework/private/focus-table at 0
KEEP framework/preferences at 0
KEEP framework/test at 0
KEEP framework/gui-utils at 0
KEEP framework/decorated-editor-snip at 0
KEEP framework/private/decorated-editor-snip at 0
BYPASS scheme/base at 1
  TO racket/base at 1
KEEP "private/scheme.rkt" at 1
> (check-requires 'openssl #:show-uses? #t)
KEEP racket/base at 0
  #%module-begin at 0 (reference)
  all-from-out at 0 (syntax-local-value disappeared-use)
  provide at 0 (reference)
  require at 0 (reference)
KEEP "mzssl.rkt" at 0
  ports->ssl-ports at 0 (provide)
  ssl-abandon-port at 0 (provide)
  ssl-accept at 0 (provide)
  ssl-accept/enable-break at 0 (provide)
  ssl-addresses at 0 (provide)
  ssl-available? at 0 (provide)
  ssl-channel-binding at 0 (provide)
  ssl-client-context? at 0 (provide)
  ssl-close at 0 (provide)
  ssl-connect at 0 (provide)
```

```
  ssl-connect/enable-break at 0 (provide)
  ssl-context? at 0 (provide)
  ssl-default-channel-binding at 0 (provide)
  ssl-default-verify-sources at 0 (provide)
  ssl-dh4096-param-bytes at 0 (provide)
  ssl-get-alpn-selected at 0 (provide)
  ssl-listen at 0 (provide)
  ssl-listener? at 0 (provide)
  ssl-load-certificate-chain! at 0 (provide)
  ssl-load-default-verify-sources! at 0 (provide)
  ssl-load-fail-reason at 0 (provide)
  ssl-load-private-key! at 0 (provide)
  ssl-load-suggested-certificate-authorities! at 0 (provide)
  ssl-load-verify-root-certificates! at 0 (provide)
  ssl-load-verify-source! at 0 (provide)
  ssl-make-client-context at 0 (provide)
  ssl-make-server-context at 0 (provide)
  ssl-max-client-protocol at 0 (provide)
  ssl-max-server-protocol at 0 (provide)
  ssl-peer-certificate-hostnames at 0 (provide)
  ssl-peer-check-hostname at 0 (provide)
  ssl-peer-issuer-name at 0 (provide)
  ssl-peer-subject-name at 0 (provide)
  ssl-peer-verified? at 0 (provide)
  ssl-port? at 0 (provide)
  ssl-protocol-symbol/c at 0 (provide)
  ssl-protocol-version at 0 (provide)
  ssl-seal-context! at 0 (provide)
  ssl-secure-client-context at 0 (provide)
  ssl-server-context-enable-dhe! at 0 (provide)
  ssl-server-context-enable-ecdhe! at 0 (provide)
  ssl-server-context? at 0 (provide)
  ssl-set-ciphers! at 0 (provide)
  ssl-set-keylogger! at 0 (provide)
  ssl-set-server-alpn! at 0 (provide)
  ssl-set-server-name-identification-callback! at 0 (provide)
  ssl-set-verify! at 0 (provide)
  ssl-set-verify-hostname! at 0 (provide)
  ssl-try-verify! at 0 (provide)
  supported-client-protocols at 0 (provide)
  supported-server-protocols at 0 (provide)
```

```
(show-requires module-name)
 → (listof (list/c 'keep   module-path? number?)
           (list/c 'bypass module-path? number? list?)
           (list/c 'drop   module-path? number?))
```

module-name : module-path?

Like check-requires, but returns the analysis as a list instead of printing it. The procedure returns one element per (non-label) require in the following format:

- (list 'keep *req-module req-phase*)
- (list 'bypass *req-module req-phase replacements*)
- (list 'drop *req-module req-phase*)

Example:

```
> (show-requires 'framework)
'((keep racket/base 0)
  (keep racket/contract/base 0)
  (keep racket/unit 0)
  (keep racket/class 0)
  (keep racket/gui/base 0)
  (keep racket/set 0)
  (keep mred/mred-unit 0)
  (keep framework/framework-unit 0)
  (keep framework/private/sig 0)
  (keep scribble/srcdoc 0)
  (keep framework/private/focus-table 0)
  (keep framework/preferences 0)
  (keep framework/test 0)
  (keep framework/gui-utils 0)
  (keep framework/decorated-editor-snip 0)
  (keep framework/private/decorated-editor-snip 0)
  (bypass scheme/base 1 ((racket/base 1 #f)))
  (keep "private/scheme.rkt" 1))
```

9 Showing Module Dependencies

```
(require macro-debugger/analysis/show-dependencies)
```
package: macro-debugger-text-lib

The "Show Dependencies" utility can be run as a raco subcommand. For example (from racket root directory):

```
raco show-dependencies -bc racket/collects/openssl/main.rkt

raco show-dependencies -c --exclude racket openssl
```

Each argument is interpreted as a file path if it exists; otherwise it is interpreted as a module path. See `show-dependencies` for a description of the output format.

```
(show-dependencies root
                   ...
                  [#:exclude exclude
                   #:exclude-deps exclude-deps
                   #:show-context? show-context?]) → void?
  root : module-path?
  exclude : (listof module-path?) = null
  exclude-deps : (listof module-path?) = null
  show-context? : boolean? = #f
```

Computes the set of modules transitively required by the *root* module(s). A *root* module is included in the output only if it is a dependency of another *root* module. The computed dependencies do not include modules reached through `dynamic-require` or `lazy-require` or referenced by `define-runtime-module-path-index` but do include modules referenced by `define-runtime-module-path` (since that implicitly creates a `for-label` dependency).

Dependencies are printed, one per line, in the following format:

```
dep-module [<- (direct-dependent ...)]
```

> Indicates that *dep-module* is transitively required by one or more *root* modules. If *show-context?* is true, then the *direct-dependent*s are shown; they are the modules reachable from (and including) the *root* modules that directly require *dep-module*.

The dependencies are trimmed by removing any module reachable from (or equal to) a module in *exclude* as well as any module reachable from (but not equal to) a module in *exclude-deps*.

Examples:

```
> (show-dependencies 'openssl
                     #:exclude (list 'racket))
ffi/file
ffi/unsafe
ffi/unsafe/alloc
ffi/unsafe/atomic
ffi/unsafe/custodian
ffi/unsafe/define
ffi/unsafe/global
ffi/unsafe/private/ffi-lib
ffi/unsafe/private/finalizer
ffi/unsafe/private/not-available
ffi/unsafe/runtime-lib
ffi/vcruntime
openssl/libcrypto
openssl/libssl
openssl/mzssl
openssl/private/ffi
racket/private/place-local
> (show-dependencies 'openssl
                     #:show-context? #t
                     #:exclude (list 'racket))
ffi/file <- (openssl/mzssl openssl/private/ffi)
ffi/unsafe <- (ffi/file ffi/unsafe/define ffi/unsafe/runtime-
lib openssl/libcrypto openssl/libssl openssl/mzssl
openssl/private/ffi)
ffi/unsafe/alloc <- (openssl/private/ffi)
ffi/unsafe/atomic <- (ffi/unsafe/alloc ffi/unsafe/custodian
openssl/mzssl openssl/private/ffi)
ffi/unsafe/custodian <- (openssl/mzssl openssl/private/ffi)
ffi/unsafe/define <- (openssl/private/ffi)
ffi/unsafe/global <- (openssl/mzssl openssl/private/ffi)
ffi/unsafe/private/ffi-lib <- (ffi/unsafe)
ffi/unsafe/private/finalizer <- (ffi/unsafe ffi/unsafe/alloc
ffi/unsafe/custodian)
ffi/unsafe/private/not-available <- (ffi/unsafe/define)
ffi/unsafe/runtime-lib <- (ffi/vcruntime)
ffi/vcruntime <- (openssl/libcrypto openssl/libssl)
openssl/libcrypto <- (openssl/libssl openssl/mzssl
openssl/private/ffi)
openssl/libssl <- (openssl/mzssl openssl/private/ffi)
openssl/mzssl <- (openssl)
openssl/private/ffi <- (openssl/mzssl)
racket/private/place-local <- (ffi/unsafe/atomic)
```

```
(get-dependencies  root
                   ...
                  [#:exclude exclude
                   #:exclude-deps exclude-deps])
 → (listof (list module-path? (listof module-path?)))
  root : module-path?
  exclude : (listof module-path?) = null
  exclude-deps : (listof module-path?) = null
```

Like show-dependencies, but returns a list instead of producing output. Each element of the list is a list containing a module path and the module paths of its immediate dependents.

Example:

```
> (get-dependencies 'openssl #:exclude (list 'racket))
'((ffi/file (openssl/mzssl openssl/private/ffi))
  (ffi/unsafe
   (ffi/file
    ffi/unsafe/define
    ffi/unsafe/runtime-lib
    openssl/libcrypto
    openssl/libssl
    openssl/mzssl
    openssl/private/ffi))
  (ffi/unsafe/alloc (openssl/private/ffi))
  (ffi/unsafe/atomic
   (ffi/unsafe/alloc ffi/unsafe/custodian openssl/mzssl
openssl/private/ffi))
  (ffi/unsafe/custodian (openssl/mzssl openssl/private/ffi))
  (ffi/unsafe/define (openssl/private/ffi))
  (ffi/unsafe/global (openssl/mzssl openssl/private/ffi))
  (ffi/unsafe/private/ffi-lib (ffi/unsafe))
  (ffi/unsafe/private/finalizer
   (ffi/unsafe ffi/unsafe/alloc ffi/unsafe/custodian))
  (ffi/unsafe/private/not-available (ffi/unsafe/define))
  (ffi/unsafe/runtime-lib (ffi/vcruntime))
  (ffi/vcruntime (openssl/libcrypto openssl/libssl))
  (openssl/libcrypto (openssl/libssl openssl/mzssl
openssl/private/ffi))
  (openssl/libssl (openssl/mzssl openssl/private/ffi))
  (openssl/mzssl (openssl))
  (openssl/private/ffi (openssl/mzssl))
  (racket/private/place-local (ffi/unsafe/atomic)))
```

10 Macro Profiler

The Macro Profiler shows what macros contribute most to the *expanded code size* of programs. Use the Macro Profiler when your program has compiled files that are larger than expected. (The Macro Profiler does not report expansion time, but expansion time is generally proportional to code size.)

`raco macro-profiler` *module-path* ...

The Macro Profiler works by expanding the files using the Macro Debugger and recording the difference in term sizes for each macro expansion step. The size of a term is computed by counting its pairs, atoms, etc.

Consider the following partial macro expansion:

```
(m (n x)) ⇒ (o (p y) (n x)) ⇒ (f (p y) (n x) z) ⇒ (f y (n x) z)
```

The *direct* cost of `m` is 6—the size of the new term `(p y)` plus one for the additional pair to include it in the `o` arguments. Likewise, the direct cost of `o` is 2. The direct cost of `p` is -4, because the macro's result is smaller than its use.

The *total* cost of a macro consists of its direct cost *plus* the costs of any macros in the code introduced by `m`, but *not* including the costs from macro arguments like `(n x)`. So the total cost of `m` is 6 + 2 - 4 = 4, because the `o` and `p` terms were introduced by `m`. In contrast, the total cost of `o` is just 2, the same as the direct cost.

Here are some known limitations:

- Term size is an imperfect proxy for compiled code size. For example, a macro might generate a large expression that it knows the compiler will turn into a small expression via constant propagation and dead code elimination (see the "Macro-Writer's Bill of Rights"). The profiler will overestimate the code-size cost of such a macro.
- The Macro Profiler uses scopes to determine what terms were introduced by a macro, so it can be confused by certain kinds of hygiene-breaking macros.
- The profiler calculates the costs of `local-expand` assuming that is used only on macro arguments, and that the result is used in the macro's result. Macros that violate this assumption will have correspondingly incorrect profile costs.