Macro Debugger

Version 4.1

August 12, 2008

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)
```

```
(expand/step stx) → (is-a/c macro-stepper<%>)
  stx : any/c
```

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

```
macro-stepper<%> : interface?
```

```
(send a-macro-stepper at-start?) → boolean?
```

```
(send a-macro-stepper at-end?) → boolean?
```

```
(send a-macro-stepper navigate-to-start) → void?
```

```
(send a-macro-stepper navigate-to-end) → void?
```

```
(send a-macro-stepper navigate-previous) → void?
```

```
(send a-macro-stepper navigate-next) → void?
```

```
(send a-macro-stepper at-top?) → boolean?
```

```
(send a-macro-stepper at-bottom?) → boolean?
```

```
(send a-macro-stepper navigate-up) → void?
```

```
(send a-macro-stepper navigate-down) → void?
```

2 Macro expansion tools

```
(require macro-debugger/expand)
```

```
(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*.

Examples:

```
> (syntax->datum
    (expand-only #'(let ([x 1] [y 2]) (or (even? x) (even? y)))
                 (list #'or)))
reference to undefined identifier: expand-only
```

Warning: because of limitations in syntax, expansion, and hiding, the resulting syntax may not evaluate to the same thing as the original syntax.

```
(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).

Examples:

```
> (syntax->datum
    (expand/hide #'(let ([x 1] [y 2]) (or (even? x) (even? y)))
                 (list #'or)))
reference to undefined identifier: expand/hide
```

Warning: because of limitations in syntax, expansion, and hiding, the resulting syntax may not evaluate to the same thing as the original syntax.

3 Macro stepper text interface

```
(require macro-debugger/stepper-text)
```

```
(expand/step-text stx [macro-policy]) → void?
  stx : any/c
  macro-policy : (or/c (-> identifier? boolean?) = null
                       (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, no macros are hidden). A list of identifiers is also accepted.

Examples:

```
> (expand/step-text #'(let ([x 1]) (even? x)))
Macro transformation
(let ((x 1)) (even? x))
  ==>
(let-values:1 (((x) 1)) (even? x))

Tag datum
(let-values:1 (((x) 1)) (even? x))
  ==>
(let-values:1 (((x) (#%datum . 1))) (even? x))

Macro transformation
(let-values:1 (((x) (#%datum . 1))) (even? x))
  ==>
(let-values:1 (((x) '1)) (even? x))

Tag application
(let-values:1 (((x) '1)) (even? x))
  ==>
(let-values:1 (((x) '1)) (#%app even? x))

Macro transformation
(let-values:1 (((x) '1)) (#%app even? x))
  ==>
(let-values:1 (((x) '1)) (#%app:2 even? x))
```

```
(stepper-text stx macro-policy) → (symbol? -> void?)
  stx : any/c
```

```
macro-policy : (or/c (-> identifier? boolean?)
                     (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.

4 Syntax browser

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

```
(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.

5 Using the macro stepper

5.1 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.

5.2 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.

6 Using the syntax browser

6.1 Selection (bold)

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).

6.2 Primary partition (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.

6.3 Secondary partitioning (highlight)

The user may select a *secondary partitioning* from a drop-down box (or in the macro stepper, 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=?`
- `module-identifier=?`
- `module-or-top-identifier=?`
- **symbolic-identifier=?**: Two identifiers are symbolic-identifier=? if discarding all lexical context information yields the same symbol.
- **same marks**: Two identifiers have the same marks if (barring nonhygienic macros) they were produced by the same macro transformation step.

- **same source module**: The bindings of the two identifiers come from definitions in the same module.
- **same nominal module**: The bindings of the two identifiers were imported into the current context by requiring the same module.

6.4 Properties

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

- Binding

If the selection is an identifier, shows the binding information associated with the syntax object.

*Note: See the warning in the section below.

For more information, look up 'identifier-binding', 'identifier-transformer-binding', and 'identifier-template-binding' in the Help Desk.

- Source

Displays source location information about the syntax object.

- Properties

Displays properties (see 'syntax-property') of the selection when it has properties it knows the keys for.

6.5 Warnings about 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.

For example, in

```
(browse-syntax #'(lambda (foo) foo))
```

the syntax browser will report that the inner 'foo' is unbound, even though in the *program* that this syntax represents, the inner 'foo' is bound to the outer 'foo'.

6.6 Notes and Limitations

The syntax browser does not have a way of extending the set of available secondary partitions.

The syntax browser does not have a way of extending the set of known properties.

The syntax browser does not preserve the distinction between parentheses and square brackets.

7 Notes for DrScheme language implementors

The macro stepper works ”out of the box” only with certain languages out of all the languages available from the DrScheme languages menu. For example, the macro stepper is disabled for the teaching languages.

An implementor of a new DrScheme language can designate their language ”macro-steppable” by overriding the ’enable-macro-stepper?’ method of their implementation of ’drscheme:language:language<%>’. The default implementation in the mixin provided by ’drscheme:language:get-default-mixin’ returns false; override this method to return true if the macro stepper button should be shown for this language.

Note: There is currently no way to customize the behavior of the macro stepper for different languages. When enabled, the macro stepper sees exactly those terms that pass through the ’current-eval’ handler.