Title

Custom macro transformers

Author

Marc Nieper-Wißkirchen

Status

This SRFI is currently in *final* status. Here is an explanation of each status that a SRFI can hold. To provide input on this SRFI, please send email to `srfi-147@srfi.schemers.org`. To subscribe to the list, follow these instructions. You can access previous messages via the mailing list archive.

- Received: 2016-12-24
- 60-day deadline: 2016-02-22
- Draft #1 published: 2015-12-24
- Draft #2 published: 2015-12-26
- Draft #3 published: 2015-12-30
- Draft #4 published: 2017-01-02
- Finalized 2017-03-07
- Revised to fix errata:
  - 2017-04-27 (Clarified use of `begin` in transformer specs.)

Abstract

Each syntax definition assigns a macro transformer to a keyword. The macro transformer is specified by a transformer spec, which is either an instance of `syntax-rules`, an existing syntactic keyword (including macro keywords and the syntactic keywords that introduce the core forms, like `lambda`, `if`, or `define`), or a use of a macro that eventually expands into an instance of `syntax-rules`. In the latter case, the keyword of macro use is called a *custom macro transformer*.

Issues

There are currently no issues.

Rationale

While the `syntax-rules`-transformer is in many cases a simple way to define new keywords in the Scheme language, sometimes more specialized macro transformers are being called for (*e.g.* the `computation-rules` of SRFI 53). The current report on the Scheme language, the R7RS,

however, does not describe a means to define custom macro transformers based on top of syntax rules.

Such a means is described in this SRFI, namely allowing macro uses that eventually expand into a transformer spec in place of transformer spec.

A simple example is given below:

```
(define-syntax syntax-rules*
  (syntax-rules ()
    ((syntax-rules* (literal ...) (pattern . templates) ...)
     (syntax-rules (literal ...) (pattern (begin . templates)) ...))
    ((syntax-rules* ellipsis (literal ...) (pattern . templates) ...)
     (syntax-rules ellipsis (literal ...) (pattern (begin . templates)) ...))))

(let-syntax
    ((foo
     (syntax-rules* ()
       ((foo a b)
        (define a 1)
        (define b 2)))))
  (foo x y)
  (list x y))            ⟹ '(1 2)
```

A fairly large use case of this SRFI is SRFI 148, which defines a custom macro transformer for hygienic composable macros.

Specification

A Scheme system supporting this SRFI should apply the requirements of this SRFI with respect to `syntax-rules` *mutatis mutandis* to all other natively provided macro transformers, *e.g.* `sc-macro-transformer` or `syntax-case`.

A Scheme system supporting this SRFI should apply the requirements of this SRFI with respect to `define-syntax`, `let-syntax`, and `letrec-syntax` *mutatis mutandis* to all other natively provided binding facilities for keywords, *e.g.* `let-syntax/splicing` or `define-syntax-parameter`.

Syntax

This SRFI extends 7.1.5 of the R7RS as follows:

```
<transformer spec>
 -> <keyword>
 -> <macro use>
 -> (begin <definition>… <transformer spec>)
```

Whenever a keyword is bound to a macro transformer, and the macro transformer is given by a transformer spec that is a macro use, the keyword is bound to the macro transformer given by the transformer spec that results from transcribing the macro use. It is an error if the macro use

does not expand into a transformer spec (but see below). In case of transformer specs that appear in the bindings of the `let-syntax` binding construct, the expansion of the transformer spec takes place in a scope between the outer scope of the binding construct and its inner scope. In case of transformer specs that appear in the bindings of the `letrec-syntax` binding constructs, the expansion of the transformer spec takes place in the scope of the bindings of the binding construct.

If the transformer spec is a *keyword*, the keyword bound to the transformer spec essentially becomes an alias to the syntactic keyword *keyword*. (*Keyword* may also be one of the core syntactic keywords like `lambda`, `if`, `define`.) The same holds if the transformer spec is a macro use that eventually expands into a (possibly empty) sequence of multiple definitions followed by the *keyword* (possibly bound by the introduced definitions).

In order to facilitate writing sophisticated custom macro transformers, it is allowed that a transformer spec expands into a sequence introduced by the keyword `begin` of multiple definitions eventually followed by a transformer spec (whose expansion may make use of the introduced definitions).

Note: This form of `begin` is a fourth form of `begin` different from the forms described in sections 4.2.3 and sections 5.6.1 of the R7RS.

Implementation

The sample implementation redefines the following identifiers exported from the `(scheme base)` library:

- `define-syntax`
- `let-syntax`
- `letrec-syntax`
- `syntax-rules`

The sample implementation provides these exports through the `(srfi 147)` library. Typical library declarations of a program or library using `(srfi 147)` are:

```
(cond-expand
  (custom-macro-transformers
   (import (scheme base)))
  (else
   (import (except (scheme base)
                   define-syntax
                   let-syntax
                   letrec-syntax
                   syntax-rules))
   (import (srfi 147))))
```

For convenience, the sample implementation provides these library declaration in the file [custom-macro-transformers.scm](#) for inclusion via `include-library-declarations`.

A Scheme system properly supporting this SRFI should ensure that the bindings of `define-syntax`, `let-syntax`, `letrec-syntax`, and `syntax-rules` exported from `(scheme base)` and `(srfi 147)` are respectively the same. Such a Scheme system should provide the feature identifier `custom-macro-transformers`. This cannot be achieved by a fully portable implementation using only the R7RS without redefining `(scheme base)`.

Acknowledgements

Credit goes to the inventors and promoters of the beautiful `syntax-rules`-macro facility, which seems to be an indispensable and defining part of contemporary Scheme.

Copyright

Editor: Arthur A. Gleckler