alexpander egg

A syntax-rules macro-expander
Extension for Chicken Scheme
Version 1.58.3

Al Petrofsky

Table of Contents

1 About this egg

1.1 Version history

`1.58.3` expander returns single form, if possible; also handles keywords, eof-object and other number type literals

`1.58.2` Simpler use in compiled code, support for some builtin syntax

`1.58` Initial release

1.2 Usage

Load this egg like so:

```
  (require-extension alexpander)
```

2 Documentation

This extension provides a syntax-rules macro system for compiled or interpreted code. It is less featureful than the syntax-case egg and only supports pure R5RS Scheme (plus a few extensions) but may be useful if you need a leightweight syntax-rules macro expander at run-time.

To use it at runtime only, evaluate `(require 'alexpander)`. To use it in the interpreter or for compiled code, add `(require-extension alexpander)` or add the `-R alexpander` option.

Implementations of the special forms `declare` and `require-extension` are available, otherwise this expander only knows about R5RS syntax.

```
EXTENSIONS:

The expander supports all the features of the r5rs macro system,
plus several extensions in the way syntaxes can be specified and
used, which are best summarized in BNF:

Modified r5rs productions:
  <expression> ---> <variable> | <literal> | <procedure call>
                  | <lambda expression> | <conditional> | <assignment>
                  | <derived expression> | <macro use> | <macro block>
                  | <keyword>
  <syntax definition> ---> (define-syntax <keyword> <syntax or expression>)
                         | (begin <syntax definition>*)
                         | <macro use>
  <syntax spec> --> (<keyword> <syntax or expression>)
  <syntax or expression> --> <syntax> | <expression>
  <macro use> ---> (<syntax> <datum>*)
  <definition> ---> (define <variable> <expression>)
                  | (define (<variable> <def formals>) <body>)
                  | (define <expression>)
                  | (begin <definition>*)
                  | <macro use>
                  | <syntax definition>
  <command or definition> ---> <command> | <definition>
                             | (begin <command or definition>*)
                             | <top-level macro block>
                             | <macro use>

New productions:
  <syntax> --> <transformer spec>
             | <keyword>
             | <macro use>
             | <syntax macro block>
  <syntax macro block> --> (<syntax-only block stuff> <syntax>)
  <top-level macro block>
```

```
      --> (<syntax-only block stuff> <command or definition>)
  <syntax-only block stuff>
     ---> <let-or-letrec-syntax> (<syntax spec>*) <syntax definition>*
  <let-or-letrec-syntax> ---> let-syntax | letrec-syntax
```

These extensions all have the obvious meaning.

Okay, I'll elaborate on that a little bit. Consider the intializer position of a syntax definition and the head position of a list-format expression:

```
  (define-syntax <keyword> <xxx>)

  (<yyy> <foo>*)
```

In r5rs, <xxx> must be a transformer. <Yyy> may be an expression, in which case the enclosing expression is taken to be a procedure call and the <foo>s are the expressions for the operands, or <yyy> may be a keyword bound to a syntax (a builtin or transformer), in which case the <foo>s are processed according to that syntax.

The core generalization in our system is that both <xxx> and <yyy> may be any type of expression or syntax. The four forms of syntax allowed are: a transformer (as allowed in the <xxx> position in r5rs), a keyword (as allowed in the <yyy> position in r5rs), a macro use that expands into a syntax, and a macro block (let-syntax or letrec-syntax) whose body is a syntax.

Some examples:

```
 ;; a macro with a local macro
 (let-syntax ((foo (let-syntax ((bar (syntax-rules () ((bar x) (- x)))))
                     (syntax-rules () ((foo) (bar 2))))))
   (foo))
 => -2

 ;; an anonymous let transformer, used directly in a macro call.
 ((syntax-rules ()
    ((_ ((var init) ...) . body)
     ((lambda (var ...) . body) init ...)))
  ((x 1) (y 2))
  (+ x y))
 => 3

 ;; a keyword used to initialize a keyword
 (let-syntax ((q quote)) (q x)) => x
 ;; Binding a keyword to an expression (which could also be thought
 ;; of as creating a macro that is called without arguments).
 (let ((n 0))
   (let-syntax ((x (set! n (+ n 1))))
     (begin x x x n)))
 => 3

 (let-syntax ((x append)) ((x x))) => ()
```

Top-level macro blocks.

At top level, if a macro block (a let-syntax or letrec-syntax form) has only one body element, that element need not be an expression (as would be required in r5rs). Instead, it may be anything allowed at top level: an expression, a definition, a begin sequence of top-level forms, or another macro block containing a top-level form.

```
(let-syntax ((- quote))
  (define x (- 1)))
(list x (- 1)) => (1 -1)
```

Note that, unlike the similar extension in Chez scheme 6.0, this is still r5rs-compatible, because we only treat definitions within the last body element as top-level definitions (and r5rs does not allow internal definitions within a body’s last element, even if it is a begin form):

```
(begin
  (define x 1)
  (let-syntax ()
    (define x 2)
    ’blah)
  x)
=> 1, in r5rs and alexpander, but 2 in Chez scheme

(begin
  (define x 1)
  (let-syntax ()
    (begin (define x 2)
           ’blah))
  x)
=> 2, in alexpander and in Chez scheme, but an error in r5rs.
```

Expressions among internal definitions.

A definition of the form (define <expression>) causes the expression to be evaluated at the conclusion of any enclosing set of internal definitons. That is, at top level, (define <expression>) is equivalent to just plain <expression>. As for internal definitions, the following are equivalent:

```
(let ()
  (define v1 <init1>)
  (define <expr1>)
  (define <expr2>)
  (define v2 <init2>)
  (define <expr3>)
  (begin
    <expr4>
    <expr5>))

(let ()
  (define v1 <init1>)
  (define v2 <init2>)
  (begin
    <expr1>
    <expr2>
    <expr3>
    <expr4>
    <expr5>))
```

(Yes, it would probably be better to have a separate builtin for this rather than to overload define.)

This feature makes it possible to implement a define-values that works properly both at top-level and among internal definitions:

```
(define define-values-temp #f)

(define-syntax define-values
  (syntax-rules ()
    ((define-values (var ...) init)
     (begin
       (define define-values-temp (call-with-values (lambda () init) list))
       (define var #f) ...
       (define
         (set!-values (var ...) (apply values define-values-temp)))))))
```

(Set!-values is implementable using just r5rs features and is left as an exercise.)

When used among internal definitions, the definition of define-values-temp in define-values's output creates a local binding, and thus the top-level binding of define-values-temp is irrelevant. When used at top-level, the definition of define-values-temp in the output does not create a binding, it mutates the top-level binding of define-values-temp. Thus, all top-level uses of define-values share a single temp variable. For internal-definition-level uses of define-values, a single shared temp would not be sufficient, but things work out okay because hygienic renaming causes each such use to create a distinct temp variable.

The version below works the same way, but hides from the top-level environment the temp that is shared by top-level uses of define-values. For a bit of tutorial and rationale about this technique, see usenet article <8765tos2y9.fsf@radish.petrofsky.org>:

```
(define-syntax define-values
  (let-syntax ((temp (syntax-rules ())))
    (syntax-rules ()
      ((define-values (var ...) init)
       (begin
         (define temp (call-with-values (lambda () init) list))
         (define var #f) ...
         (define (set!-values (var ...) (apply values temp)))))))
```

Internal syntax definitions.

Internal syntax definitions are supported wherever they would make sense (see the BNF) and have the letrec-syntax semantics you would expect. It is legal for the initializer of an internal variable definition to use one of the internal syntax definitions in the same body:

```
(let ()
  (define x (y))
  (define-syntax y (syntax-rules () ((y) 1)))
  x)
=> 1
```

It's also legal for internal syntax definitions to be mutually recursive transformers, but it is an error for the expansion of a syntax definition's initializer to require the result of another initializer:

```
(let ()
  (define-syntax m1 (syntax-rules () ((m1) #f) ((m1 . args) (m2 . args))))
  (define-syntax m2 (syntax-rules () ((m2 arg . args) (m1 . args))))
  (m1 foo bar baz))
=> #f

(let ()
  (define-syntax simple-transformer
    (syntax-rules ()
      ((simple-transformer pattern template)
       (syntax-rules () (pattern template)))))
  (define-syntax m (simple-transformer (m x) (- x)))
  (m 1))
=> error ("Premature use of keyword bound by an internal define-syntax")

(let ()
  (define-syntax simple-transformer
    (syntax-rules ()
      ((simple-transformer pattern template)
       (syntax-rules () (pattern template)))))
  (let ()
    (define-syntax m (simple-transformer (m x) (- x)))
    (m 1)))
=> -1
```

Syntax-rules ellipsis

Per draft SRFI-46, syntax-rules transformers can specify the identifier to be used as the ellipsis (such a specification is treated as a hygienic binding), and a list pattern may contain subpatterns after an ellipsis as well as before it:

```
  <transformer spec> ---> (syntax-rules (<identifier>*) <syntax rule>*)
             | (syntax-rules <ellipsis> (<identifier>*) <syntax rule>*)

  <syntax rule> ---> (<pattern> <template>)

  <pattern> ---> <pattern identifier>
               | (<pattern>*)
               | (<pattern>+ . <pattern>)
               | (<pattern>* <pattern> <ellipsis> <pattern>*)
               | #(<pattern>*)
               | #(<pattern>* <pattern> <ellipsis> <pattern>*)
               | <pattern datum>

  <pattern identifier> ---> <identifier>
  <ellipsis> ---> <identifier>
```

Improved nested unquote-splicing.

Quasiquote is extended to make commas and comma-ats distributive
over a nested comma-at, as in Common Lisp’s backquote. See my
2004-09-03 usenet article <87pt53f9f2.fsf@radish.petrofsky.org>,
Bawden’s 1999 quasiquotation paper, and Appendix C of Steele’s
"Common Lisp the Language 2nd edition".

```
  <splicing unquotation 1> ---> ,@<qq template 0>
                              | (unquote-splicing <qq template 0>)

  <splicing unquotation D> ---> ,@<qq template D-1>
                              | ,<splicing unquotaion D-1>
                              | ,@<splicing unquotaion D-1>
                              | (unquote-splicing <qq template D-1>)
                              | (unquote <splicing unquotaion D-1>)
                              | (unquote-splicing <splicing unquotaion D-1>)
```

When a comma at-sign and the expression that follows it are being
replaced by the elements of the list that resulted from the
expression’s evaluation, any sequence of commas and comma at-signs
that immediately preceeded the comma at-sign is also removed and is
added to the front of each of the replacements.

```
 (let ((x ’(a b c))) ‘‘(,,x ,@,x ,,@x ,@,@x))
 => ‘(,(a b c) ,@(a b c) ,a ,b ,c ,@a ,@b ,@c)

 ‘‘(,,@’() ,@,@(list))
 => ‘()

 ‘‘‘‘‘(a ,(b c ,@,,@,@(list ’a ’b ’c)))
 => ‘‘‘‘(a ,(b c ,@,,@a ,@,,@b ,@,,@c))

(let ((vars ’(x y)))
  (eval ‘(let ((x ’(1 2)) (y ’(3 4)))
           ‘(foo ,@,@vars))
        (null-environment 5)))
=> (foo 1 2 3 4)
```

3 License

Index

(Index is nonexistent)