Title

Extended LET-syntax for multiple values

Author

Sebastian Egner

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-71@srfi.schemers.org`. To subscribe to the list, follow these instructions. You can access previous messages via the mailing list archive.

- Received: 2005-05-16
- Draft: 2005-05-16--2005-07-14
- Revised: 2005-05-18
- Revised: 2005-08-01
- Final: 2005-08-12

Abstract

This SRFI is a proposal for extending `let`, `let*`, and `letrec` for receiving multiple values. The syntactic extension is fully compatible with the existing syntax. It is the intention that single-value bindings, i.e. `(let ((var expr)) ...)`, and multiple-value binding can be mixed freely and conveniently.

The most simple form of the new syntax is best explained by an example:

```
(define (quo-rem x y)
  (values (quotient x y) (remainder x y)))

(define (quo x y)
  (let ((q r (quo-rem x y)))
    q))
```

The procedure `quo-rem` delivers two values to its continuation. These values are received as `q` and `r` in the `let`-expression of the procedure `quo`. In other words, the syntax of `let` is extended such that several variables can be specified---and these variables receive the values delivered by the expression `(quo-rem x y)`.

The syntax of `let` is further extended to cases in which a rest argument receives the list of all residual values. Again by example,

```
(let (((values y1 y2 . y3+) (foo x)))
   body)
```

In this example, `values` is a syntactic keyword indicating the presence of multiple values to be received, and `y1`, `y2`, and `y3+`, resp., are variables bound to the first value, the second value, and the list of the remaining values, resp., as produced by `(foo x)`. The syntactic keyword `values` allows receiving all values as in `(let (((values . xs) (foo x))) body)`. It also allows receiving no values at all as in `(let (((values) (for-each foo list))) body)`.

A common application of binding multiple values is decomposing data structures into their components. This mechanism is illustrated in its most primitive form as follows: The procedure `uncons` (defined below) decomposes a pair `x` into its car and its cdr and delivers them as two values to its continuation. Then an extended `let` can receive these values:

```
(let ((car-x cdr-x (uncons x)))
  (foo car-x cdr-x))
```

Of course, for pairs this method is probably neither faster nor clearer than using the procedures `car` and `cdr`. However, for data structures doing substantial work upon decomposition this is different: Extracting the element of highest priority from a priority queue, while at the same time constructing the residual queue, can both be more efficient and more convenient than doing both operations independently. In fact, the `quo-rem` example illustrates this point already as both quotient and remainder are probably computed by a common exact division algorithm. (And often caching is used to avoid executing this algorithm twice as often as needed.)

As the last feature of this SRFI, a mechanism is specified to store multiple values in heap-allocated data structures. For this purpose, `values->list` and `values->vector` construct a list (a vector, resp.) storing all values delivered by evaluating their argument expression. Note that these operations cannot be procedures.

Rationale

My original motivation for writing this SRFI is my unhappiness with the current state of affairs in Scheme with respect to multiple values. Multiple values are mandatory in the Revised^5 Report on the Algorithmic Language Scheme (R5RS), and they are fully available in all major Scheme implementations. Yet there is often a painful hesitation about using them.

The reason for this hesitation is that multiple values are nearly fully integrated into Scheme---but not quite. (Unlike for example in the languages Matlab, Octave, or the computer algebra system Magma, in which returning multiple values from a function is the most natural thing in the world: `q, r := quo_rem(x, y);`) However, considerable progress has been made on this point, and I understand this SRFI as a minor contribution "placing the last corner stone". But first a very brief history of multiple values in Scheme, as far as relevant for this SRFI.

R5RS specifies the procedures `values` and `call-with-values` for passing any number of values from a producer procedure to a consumer procedure. This is the only construct in R5RS dealing with multiple values explicitly, and it is sufficient to implement anything that can be implemented for multiple values.

However, as John David Stone observed in SRFI 8, the mechanism exposes explicitly how multiple values are passed---but that is hardly ever interesting. In fact, `call-with-values` is often clumsy because the continuations are made explicit. SRFI 8 improves on this situation by adding the special form `(receive <formals> <expression> <body>)` for receiving several values produced by the expression in variables specified by `<formals>` and using them in the body.

The major limitation of `receive` is that it can only handle a single expression, which means programs dealing with multiple values frequently get deeply nested. SRFI 11 provides a more versatile construct: `Let-values` and `let*-values` are modeled after `let` and `let*` but replace `<variable>` by an argument list with the syntax of `<formals>`. The `let-values` binding construct makes multiple values about as convenient as it will ever get in Scheme. Its primary shortcoming is that `let-values` is incompatible with the existing syntax of `let`: In `(let-values ((v x)) ...)` is `v` bound to the list of all values delivered by `x` (as in SRFI 11)? Or is `x` to deliver a single value to be bound to `v` (as in `let`)? Refer to the discussion archive of SRFI 11 for details. Moreover, `let-values` suffers from "parenthesis complexity", despite Scheme programmers are tolerant to braces.

Eli Barzilay's Swindle library (for MzScheme) on the other hand redefines let to include multiple-values and internal procedures: The syntactic keyword `values` indicates the presence of multiple values, while additional parentheses (with the syntax of `<formals>`) indicate a `lambda`-expression as right-hand side.

This SRFI follows Eli's approach, while keeping the syntax simple (few parentheses and concepts) and adding tools for dealing more conveniently with multiple values. The aim is convenient integration of multiple values into Scheme, at full coexistence with the existing syntax (R5RS.) This is achieved by extending the syntax in two different ways (multiple left-hand sides or a syntactic keyword) and adding operations to convert between (implicitly passed) values and (first class) data structures.

Finally, I would like to mention that Oscar Waddell et al. describe an efficient compilation method for Scheme's `letrec` (Fixing Letrec) and propose a `letrec*` binding construct to as a basis for internal `define`. I expect their compilation method (and `letrec*`) and this SRFI to be fully compatible with one another, although I have not checked this claim by way of implementation.

Specification

The syntax of Scheme (R5RS, Section 7.1.3.) is extended by replacing the existing production:

```
<binding spec> --> (<variable> <expression>)
```

by the three new productions

```
<binding spec> --> ((values <variable>*) <expression>)
<binding spec> --> ((values <variable>* . <variable>) <expression>)
<binding spec> --> (<variable>+ <expression>)
```

The form `(<variable>+ <expression>)` is just an abbreviation for `((values <variable>+) <expression>)`, and it includes the original `<binding spec>` of R5RS.

The first two forms are evaluated as follows: The variables are bound and the expression is evaluated according to the enclosing construct (either `let`, `let*`, or `letrec`.) However, the expression may deliver any number of values to its continuation, which stores these values into the variables specified, possibly allocating a rest list in case of the `. <variable>` form.

The number of values delivered by the expression must match the number of values expected by the binding specification. Otherwise an error is raised, as `call-with-values` would. This implies in particular, that each binding of a named let involves exactly one value, because this binding can also be an argument to a lambda-expression.

Standard operations

The following procedures, specified in terms of standard procedures, are added to the set of standard procedures:

```
(define (uncons pair)
  (values (car pair) (cdr pair)))

(define (uncons-2 list)
  (values (car list) (cadr list) (cddr list)))

(define (uncons-3 list)
  (values (car list) (cadr list) (caddr list) (cdddr list)))

(define (uncons-4 list)
  (values (car list) (cadr list) (caddr list) (cadddr list) (cddddr list)))

(define (uncons-cons alist)
  (values (caar alist) (cdar alist) (cdr alist)))

(define (unlist list)
  (apply values list))

(define (unvector vector)
  (apply values (vector->list vector)))
```

These procedures decompose the standard concrete data structures (pair, list, vector) and deliver the components as values. It is an error if the argument cannot be decomposed as expected. Note that the procedures are not necessarily implemented by the definition given above.

The preferred way of decomposing a list into the first two elements and the rest list is `(let ((x1 x2 x3+ (uncons-2 x))) body)`, and similar for three or four elements and a rest. This is *not* equivalent to `(let (((values x1 x2 . x3+) (unlist x))) body)` because the latter binds `x3+` to a newly allocated copy of `(cddr x)`.

Finally, the following two macros are added to the standard macros:

```
(values->list   <expression>)
(values->vector <expression>)
```

These operation receive all values (if any) delivered by their argument expression and return a newly allocated list (vector, resp.) of these values. Note that `values->list` is *not* the same as `list` (the procedure returning the list of its arguments).

Design Rationale

Which alternatives designs for the syntax were considered?

This SRFI defines two notations for receiving several values: Using the keyword `values`, or simply listing the variables if there is at least one. There are several alternatives for this design, some of which were proposed during the discussion. (Refer in particular to msg00000, msg00001, and msg00002, msg00007.) The alternatives considered include:

1. Just listing the variables (no syntactic keyword at all) as in `(let ((x1 x2 expr)) body)`.
2. Only using a keyword, e.g. `values`, as in `(let (((values x1 x2 . x3+) expr)) body)`.
3. A keyword, e.g. `dot`, indicating the rest list as in `(let ((x1 x2 dot x3+ expr)) body)`.
4. The keyword `...` indicating the rest list as in `(let ((x1 x2 x3+ ... expr)) body)`.
5. A keyword, e.g. `rest`, indicating the rest list as in `(let ((x1 x2 (rest x3+) expr)) body)`.
6. Using the `<formals>` syntax of R5RS as in `(let (((x1 x2 . x3+) expr)) body)`.
7. Mimicking `<formals>` but with one level of parentheses removed as in `(let ((x1 x2 . x3+ expr)) body)`.
8. As the previous but with additional syntax for "no values" and "just a rest", e.g. `(let ((! expr)) body)` and `(let ((xs . expr)) body)`.

The requirements for the design are compatibility with the existing `let`, concise notation for the frequent use cases, robustness against most common mistakes, and full flexibility of receiving values.

For the sake of compatibility, only modifications of `<binding spec>` were considered. The alternative, i.e. modifying the syntax of `let` in other ways, was not considered. Concerning concise notation, by far the most convenient notation is *listing the variables*. As this notation also covers the existing syntax, it was adopted as the basis of the extension to be specified.

The *listing the variables* notation is limited by the fact that the preferred marker for a rest list (".") cannot follow an opening parenthesis as in `(let ((. xs expr)) body)`, nor that it can be followed by two syntactic elements as in `(let ((x1 . x2+ expr)) body)`. Lifting these restrictions would require major modifications in unrelated parts of the Scheme syntax, which is not an attractive option.

Another problematic aspect of the *listing the variables* notation is the case of no variables at all. While this case is not conflicting with existing syntax, it seriously harms syntactic robustness: `(let (((run! foo))) body)` and `(let ((run! foo)) body)` would both be syntactically correct and could easily be confused with one another. For this reason, the notation of listing the variables was restricted to one or more variables.

This leaves the problem of extending the notation in order to cover rest arguments and the "no values"-case. This can either be done *ad hoc*, covering the open cases, or by adding a general notation covering all cases. In view of readability and uniformity (useful when code gets processed automatically) the latter approach was chosen. This has resulted in the design specified in this SRFI.

Why is `values` needed in the "zero values"-case?

The syntax specified in this SRFI allows zero variables being bound in a binding specification using the syntax `(let (((values) (for-each foo (bar)))) body)`. An alternative is allowing `(<expression>)` as a binding specification. (Refer to the discussion archive starting at msg00001.)

The syntax specified in this SRFI is designed for static detection of the most frequent types (forgotten parentheses). For example, writing `(let ((values) (for-each foo (bar))) body)` is not a well-formed let-expression in this SRFI.

In the alternative syntax, both `(let (((for-each foo (bar)))) body)` and `(let ((for-each foo (bar))) body)` are syntactically correct. The first just executes `(for-each foo (bar))`, whereas the second binds the variables `for-each` and `foo` to the two values delivered by `(bar)`. Assuming the first meaning was intended, the error will probably manifest

itself at the moment `(bar)` fails to deliver exactly two values. Unless it does, in which case the error must manifest itself much further downstream from the fact that `foo` never got called.

In order to avoid this sort of expensive typos, the syntax proposed in this SRFI is more verbose than it needs to be.

Why not also include a syntax for procedures?

This SRFI is a proposal for extending the syntax of `let` etc. in order to include multiple values. It is also desirable to extend the syntax of `let` for simplifying the definition of local procedures. (For example, as in Swindle.) However, this SRFI does not include this feature.

The reason I have chosen not restrict this SRFI to a syntax for multiple values is simplicity.

Why the names `unlist` etc.?

An alternative naming convention for the decomposition operation `unlist` is `list->values`, which is more symmetric with respect to its inverse operation `values->list`.

This symmetry ends, however, as soon as more complicated data structures with other operations are involved. Then it becomes apparent that the same data structure can support different decomposition operations: A double-ended queue (deque) for example supports splitting off the head and splitting of the tail; and neither of these operations should be named `deque->values`. The `un`-convention covers this in a natural way.

Please also refer to the double-ended queue (deque) example in examples.scm to see how to use decomposition procedures for dealing with data structures.

Which decomposition operations are included?

The particular set of operations specified in this SRFI for decomposing lists represents a trade-off between limiting the number of operations and convenience.

As Al Petrofsky has pointed out during the discussion (msg00018) it is not sufficient to have only `unlist` as this will copy the rest list. For this reason specialized decomposition operations for splitting off the first 1, ..., 4 elements are provided, and a decomposition operation expecting the first element to be a pair itself. These appear to be the most common cases.

Implementation

The reference implementation is written in R5RS using hygienic macros, only. It is not possible, however, to portably detect read access to an uninitialized variable introduced by `letrec`. The definition of the actual functionality can be found here. The implementation defines macros

`srfi-let/*/rec` etc. in terms of `r5rs-let/*/rec`. Implementors may use this to redefine (or even re-implement) `let/*/rec` in terms of `srfi-let/*/rec`, while providing implementations of `r5rs-let/*/rec`. An efficient method for the latter is given in Fixing Letrec by O. Waddell et al.

R5RS: For trying out the functionality, a complete implementation under R5RS can be found here. It defines `r5rs-let/*/rec` in terms of `lambda` and redefines `let/*/rec` as `srfi-let/*/rec`. This may not be the most efficient implementation, because many Scheme systems handle `let` etc. specially and do not reduce it into `lambda`

PLT 208: The implementation found here uses PLT's module system for exporting `srfi-let/*/rec` under the name of `let/*/rec`, while defining `r5rs-let/*/rec` as a copy of the built-in `let/*/rec`. This code should be efficient.

Examples using the new functionality can be found in examples.scm.

References

[R5RS]	Richard Kelsey, William Clinger, and Jonathan Rees (eds.): Revised^5 Report on the Algorithmic Language Scheme of 20 February 1998. Higher-Order and Symbolic Computation, Vol. 11, No. 1, September 1998. http://schemers.org/Documents/Standards/R5RS/.
[SRFI 8]	John David Stone: `Receive`: Binding to multiple values. http://srfi.schemers.org/srfi-8/
[SRFI 11]	Lars T. Hansen: Syntax for receiving multiple values. http://srfi.schemers.org/srfi-11/
[Swindle]	Eli Barzilay: Swindle, documentation for "base.ss" (Swindle Version 20040908.) http://www.cs.cornell.edu/eli/Swindle/base-doc.html#let
[Fix]	O. Waddell, D. Sarkar, R. K. Dybvig: Fixing Letrec: A Faithful Yet Efficient Implementation of Scheme's Recursive Binding Construct. To appear, 2005. http://www.cs.indiana.edu/~dyb/pubs/fixing-letrec.pdf

Copyright

Author: *Sebastian Egner*
Editor: *Mike Sperber*
Last modified: Sun Jan 28 13:40:16 MET 2007