comp.lang.scheme

An Advanced Syntax-Rules Primer for the Mildly Insane

Al Petrofsky <a...@petrofsky.org>

g...@mo.msk.ru (Golubev I. N.) writes:

> Widely known descriptions of `syntax-rules' syntax in r5rs and "The
> Scheme Programming Language, 2nd Edition" seem unsufficient to me.
> They do not tell explicitly just how and when syntax expander
> determines that "literal identifier is inserted as a bound
> identitifer" so that it must be "renamed to prevent inadvertent
> captures of free identifiers".

That's because it's a difficult thing to explicitly tell.
Syntax-rules is quite easy to use and sufficiently powerful for many
tasks without a deep understanding of its exact semantics. It is
intended to more often create a correct macro from the input of a
naive user than does a traditional lisp macro system. If you have
read r5rs and understand it well enough to write a cond macro
(including =>), and you're too foolhardy to realize that you don't
really want to know enough detail about syntax-rules to truly
understand the letrec macro, then read on...

============================== ==============================
==========

The r5rs macro system documentation is hard to fully understand, and
is in some aspects incomplete. Compared to the other features in
r5rs, the macros were based on more recent research, and were added to
the report at a point when the rnrs revision process was stalled. I
think perhaps the only people who read and deeply understood the
syntax-rules specification and contributed to its revision were people
who had experience in writing a lexically-scoped macro system and were
so familiar with certain aspects of such systems that they failed to
notice that these aspects weren't adequately defined in the
specification. At the other extreme, some of the issues with macro
uses that expand into macro transformers were not yet fully understood
by anyone, and this also led to deficiencies in the specification.

As a result, I think it is virtually impossible to determine the exact
intended semantics solely from r5rs. So the short answer to your
questions is: instead of trying to figure out why it follows from the
syntax-rules specification that the letrec macro works, try to figure
out what the syntax-rules specification must have been trying to say,
given that it must have intended to specify a semantics under which
the letrec macro works.

Below I will try to address some of the weaknesses of the spec. This
article might not be any more accessible than r5rs, but I hope to
illuminate some of the trickier aspects of lexically-scoped macros
that r5rs barely addresses.

First note that the definition of a "literal identifier" in a pattern
or template is "any identifier in the pattern or template that is not
a pattern variable or ellipsis". However, the semantics of literal
identifiers in the templates are unrelated to the semantics of literal
identifiers in the patterns.

Here is the key paragraph in R5RS:

> Identifiers that appear in the template but are not pattern
> variables or the identifier ... are inserted into the output as
> literal identifiers. If a literal identifier is inserted as a free
> identifier then it refers to the binding of that identifier within
> whose scope the instance of syntax-rules appears. If a literal
> identifier is inserted as a bound identifier then it is in effect
> renamed to prevent inadvertent captures of free identifiers.

It doesn't really make sense to say "is inserted as a free identifier"
or "is inserted as a bound identifier", because when the identifier is
inserted, it is neither free nor bound. In fact, it may be
subsequently replicated, and in the final resulting expression, some
of the copies could be free uses, some bound uses, and still other
copies could become bindings (which are distinct from bound uses) and
literals. For example:

```
  (let ((a 1))
    (letrec-syntax
        ((foo (syntax-rules ()
                ((_ b)
                 (bar a b))))
         (bar (syntax-rules ()
                ((_ c d)
                 (cons c (let ((c 3))
                           (list d c 'c)))))))
      (let ((a 2))
        (foo a))))

  => (1 2 3 a)
```

When the use of foo is transcribed, the a in the template (bar a b) is
inserted into the output as a literal identifier. When the output is
then transcribed according to bar, this identifier is replicated and
used to replace the four occurrences of the pattern variable c in the
template (cons c (let ((c 3)) (list d c 'c))). The completely
macro-expanded version of the expression looks like this:

```
  (let ((a 1))
    (let ((a 2))
      (cons a (let ((a 3))
                (list a a 'a)))))
  => (1 2 3 a)
```

Why doesn't this evaluate to (2 3 3 a)? Because macros introduce new
aspects to lexical scoping, such that the meaning of an identifer in a
macro-expanded expression depends not only on the name of the
identifier used and on which bindings' regions enclose it, but also on
when the identifier was inserted, and on which bindings' regions
enclose the macro that did the insertion. Specifically, two
identifiers are only considered to be entirely equivalent if they have

the same name, and either they were both part of the original program text, or they were inserted by the same template transcription and the corresponding literal identifiers in the template were entirely equivalent. If an identifier use is in the region of one or more bindings for entirely equivalent identifiers, then the identifier refers to the innermost such binding. If there is no such binding, then an identifier use refers to whatever binding is referred to by the corresponding literal identifier in the template whose transcription inserted it. An identifier use is an error if no such binding exists. When we get to macro uses that expand into macro transformers, it will become important that these rules are recursive.

Note that I said "inserted by the same template transcription" rather than "inserted by the same template": each time a template that contains a literal identifier is transcribed, it inserts a new distinct copy of that identifier.

Now let's walk through the expansion of that example, and indicate the uniqueness of a macro-inserted identifier by suffixing its name with #n, where n is the serial number of the template transcription that inserted the identifier. This will make it easy to determine if two identifiers are exactly equivalent.

```
  (let ((a 1))
    (letrec-syntax
        ((foo (syntax-rules ()
                ((_ b)
                 (bar a b))))
         (bar (syntax-rules ()
                ((_ c d)
                 (cons c (let ((c 3))
                           (list d c (quote c))))))))
      (let ((a 2))
        (foo a))))
```

The use of foo at the bottom is in the region of the letrec-syntax binding for foo, and the foo identifiers exactly match (neither was inserted by a template transcription), so the foo transformer is applied, the pattern variable b is matched to a, and the template (bar a b) is transcribed: the pattern variable b is replaced with the part of the input that matched it, a, and the literal identifiers bar and a are replaced by freshly-inserted identifiers with the same names but new insertion serial numbers. That gives us (bar#1 a#1 a). The whole expression is now:

```
  (let ((a 1))
    (letrec-syntax
        ((foo (syntax-rules ()
                ((_ b)
                 (bar a b))))
         (bar (syntax-rules ()
                ((_ c d)
                 (cons c (let ((c 3))
                           (list d c 'c)))))))
      (let ((a 2))
        (bar#1 a#1 a))))
```

The expression in the last line has a use of bar#1 that is not in the

region of any binding for bar#1. We therefore go to the template that
inserted bar#1 (i.e. the template used in transcription serial number
one), which is (bar a b), and look for a binding for bar whose region
encloses the template. We find the letrec-syntax binding for bar, so
that transformer is applied, the pattern variables c and d are matched
to a#1 and a respectively, and the template is transcribed:

```
  (let ((a 1))
    (letrec-syntax
       ((foo (syntax-rules ()
               ((_ b)
                (bar a b))))
        (bar (syntax-rules ()
               ((_ c d)
                (cons c (let ((c 3))
                          (list d c 'c)))))))
      (let ((a 2))
        (cons#2 a#1 (let#2 ((a#1 3))
                   (list#2 a a#1 (quote#2 a#1)))))))
```

Now we have a use of cons#2 with no binding of cons#2. We therefore
go to the template of transcription number two, and from there the
nearest binding of cons is the ordinary top-level one. Similarly,
let#2, list#2, and quote#2 all refer to ordinary let, list, and quote.
(We'll assume that let is a builtin rather than a macro.) The use of
a#1 as the first argument to cons is not in the region of any binding
for a#1, so we go to the template of transcription number one, which
was (bar a b), and look for a binding for a whose region encloses the
template. Notice that the program contains two bindings for a, and
the cons expression is in the regions of both of them, but the
template is only in the region of the outer one, so that is the one
that is used, and its value, 1, is passed as the first argument to
cons. The use of a in the last line is inside the regions of two
bindings for a, and the innermost one holds the value 2, so 2 is
passed as the first argument to list. The a#1 refers to the only
binding for a#1, which contains the value 3. The quoted identifier
a#1 generates the symbol a, because the #1 is not part of the name of
the identifier, it is just extra information about the identifier that
we happen to be writing down next to its name.

Are you still with me? Anyone? Anyone?

Okay, now we get to the tricky part: macro uses that expand into macro
transformers. This means that some identifiers will have been
inserted by templates that were inserted by other templates, and the
rules for finding their bindings will need to be applied recursively.
Consider:

```
  (let ((x 1))
    (let-syntax
       ((foo (syntax-rules ()
               ((_ y) (let-syntax
                         ((bar (syntax-rules ()
                                 ((_) (let ((x 2)) y)))))
                       (bar))))))
      (foo x)))
```

The first step is straghtforward:

```
 (let ((x 1))
   (let-syntax
       ((foo (syntax-rules ()
               ((_ y) (let-syntax
                          ((bar (syntax-rules ()
                                  ((_) (let ((x 2)) y)))))
                        (bar))))))
     (let-syntax#1
         ((bar#1 (syntax-rules#1 ()
                   ((_#1) (let#1 ((x#1 2)) x)))))
       (bar#1))))
```

There is no binding for let-syntax#1, so we look for a binding of
let-syntax at the site of the template of the first macro
transcription. The nearest (and only) binding is the top-level
builtin. Next we expand the bar#1 macro use. That's interesting
because we have a template that contains two non-equivalent literal
identifiers named x: plain x and x#1.

```
 (let ((x 1))
   (let-syntax
       ((foo (syntax-rules ()
               ((_ y) (let-syntax
                          ((bar (syntax-rules ()
                                  ((_) (let ((x 2)) y)))))
                        (bar))))))
     (let-syntax#1
         ((bar#1 (syntax-rules#1 ()
                   ((_#1) (let#1 ((x#1 2)) x)))))
       (let#1#2 ((x#1#2 2)) x#2))))
```

Now it takes two steps to resolve the identifier let#1#2 back to the
ordinary top-level let. (We first look for a binding of let#1#2
starting from its use, then for a binding of let#1 starting from the
template used in macro transcription number two, and finally we look
for a binding of let starting from the template used in macro
transcription number one.) Lastly, we look for a binding for the x#2
at the end of the expression. There is none for x#2, so we go to the
template of transcription number two (the bar#1 template) and look for
a binding for x. We find the binding with value 1, and that is the
value of this whole expression.

Some scheme implementations get this example wrong and return 2
because they compare identifiers only by comparing their names and the
times of their final insertion, so the binding for x#1#2 captures the
use of x#2. This is wrong because x#1 and x are clearly distinct, and
there is no justification for a second "renaming" to cause them to
become equal again, when the whole purpose of identifiers being "in
effect renamed" is "to prevent inadvertent captures".

Here's another problem case:

```
(let ((x 1))
  (let-syntax
      ((foo (syntax-rules ()
              ((_ y) (let-syntax
                         ((bar (syntax-rules ()
                             ((_ x) y))))
                      (bar 2))))))
      (foo x)))
```

After transcribing the (foo x) macro use, we get:

```
  (let ((x 1))
    (let-syntax
        ((foo (syntax-rules ()
               ((_ y) (let-syntax
                         ((bar (syntax-rules ()
                                 ((_ x) y))))
                       (bar 2))))))
      (let-syntax#1
         ((bar#1 (syntax-rules#1 ()
                  ((_#1 x#1) x))))
        (bar#1 2))))
```

Now we have to answer this question: in the rule ((_#1 x#1) x), is the x in the template a pattern variable, or a literal identifier? I believe it should be considered distinct from the x#1 in the pattern, and that it is therefore a literal identifier (and the value of the above expression is 2), but r5rs is unclear about this, because it says that renaming applies only to "a binding for an identifier (variable or keyword)" (r5rs 4.3). Possibly the word "variable" here was meant to include "pattern variable", but nowhere else does the report use the word "binding" to refer to a pattern variable binding.

The biggest shortcoming of r5rs syntax-rules is the inability to use an ellipsis (the identifier "...") in a syntax-rules form inside of a template, because the outer syntax-rules insists on treating it specially. The amusing thing about this is that it is essentially a form of unhygiene, and could easily have been avoided by having the identifier for the ellipsis be passed in as a parameter, rather than implicitly reserving "...". So the syntax-rules design loses as a result of failing to observe the very sort of constraint that it imposes on anyone trying to write a block-with-implicitly-bound-br eak macro.

Lest I sound like I am a detractor, let me say that despite the flaws and the hairiness when you look too closely, r5rs macros were an amazing leap forward for the writing of simple macros, and the great majority of macros are simple. They're also a leap forward in providing a painful but powerful way for writing complex macros, without the scheme system having to deal with the anomalies of a "low-level" macro system.

For further reading, you may wish to find some of the papers referenced in r5rs. Look at library.readscheme.org for some of these, and for some more recent papers. In particular, read the papers about syntax-case and the chez scheme module system to learn about the most significant advances in scheme syntax since syntax-rules. There are also several good bits of macro knowledge in the scheme faq at schemers.org. To fully come to grips with the tricky details, there is no substitute for writing a simple implementation of eval, null-environment, and scheme-report-environment in scheme.

If this article actually leaves you wanting more of the same, you could try doing a search for petrofsky or oleg and "syntax-rules" on groups.google.com, which will retrieve many simple and complex examples, and a few explorations so mind-numbingly esoteric that they are must-haves in any "Syntax-Rules Syllabus for the Criminally Insane".

-al