A Syntactic Closures Macro Facility

by Chris Hanson

9 November 1991

This document describes *syntactic closures*, a low-level macro facility for the Scheme programming language. The facility is an alternative to the low-level macro facility described in the *Revised^4 Report on Scheme.* This document is an addendum to that report.

The syntactic closures facility extends the BNF rule for <transformer spec> to allow a new keyword that introduces a low-level macro transformer:

<transformer spec> ↦ `(transformer` <expression>`)`

Additionally the following procedures are added:

```
make-syntactic-closure
capture-syntactic-environment
identifier?
identifier=?
```

The description of the facility is divided into three parts. The first part defines basic terminology. The second part describes how macro transformers are defined. The third part describes the use of *identifiers*, which extend the syntactic closure mechanism to be compatible with `syntax-rules`.

Terminology

This section defines the concepts and data types used by the syntactic closures facility.

- *Forms* are the syntactic entities out of which programs are recursively constructed. A form is any expression, any definition, any syntactic keyword, or any syntactic closure. The variable name that appears in a `set!` special form is also a form. Examples of forms:

```
17
#t
car
(+ x 4)
(lambda (x) x)
(define pi 3.14159)
if
define
```

- An *alias* is an alternate name for a given symbol. It can appear anywhere in a form that the symbol could be used, and when quoted it is replaced by the symbol; however, it does not satisfy the predicate `symbol?`. Macro transformers rarely distinguish symbols from aliases, referring to both as *identifiers*.
- A *syntactic environment* maps identifiers to their meanings. More precisely, it determines whether an identifier is a syntactic keyword or a variable. If it is a keyword, the meaning is an interpretation for the form in which that keyword appears. If it is a variable, the meaning identifies which binding of that variable is referenced. In short, syntactic environments contain all of the contextual information necessary for interpreting the meaning of a particular form.
- A *syntactic closure* consists of a form, a syntactic environment, and a list of identifiers. All identifiers in the form take their meaning from the syntactic environment, except those in the given list. The identifiers in the list are to have their meanings determined later.

  A syntactic closure may be used in any context in which its form could have been used. Since a syntactic closure is also a form, it may not be used in contexts where a form would be illegal. For example, a form may not appear as a clause in the `cond` special form.

  A syntactic closure appearing in a quoted structure is replaced by its form.

Transformer Definition

This section describes the `transformer` special form and the procedures `make-syntactic-closure` and `capture-syntactic-environment`.

transformer *expression* syntax

Syntax: It is an error if this syntax occurs except as a <transformer spec>.

Semantics: The *expression* is evaluated in the standard transformer environment to yield a macro transformer as described below. This macro transformer is bound to a macro keyword by the special form in which the `transformer` expression appears (for example, `let-syntax`).

A *macro transformer* is a procedure that takes two arguments, a form and a syntactic environment, and returns a new form. The first argument, the *input form*, is the form in which the macro keyword occurred. The second argument, the *usage environment*, is the syntactic environment in which the input form occurred. The result of the transformer, the *output form*, is automatically closed in the *transformer environment*, which is the syntactic environment in which the `transformer` expression occurred.

For example, here is a definition of a `push` macro using `syntax-rules`:

```
(define-syntax push
  (syntax-rules ()
    ((push item list)
     (set! list (cons item list)))))
```

Here is an equivalent definition using `transformer`:

```
(define-syntax push
  (transformer
   (lambda (exp env)
     (let ((item
             (make-syntactic-closure env '() (cadr exp)))
            (list
             (make-syntactic-closure env '() (caddr exp))))
       `(set! ,list (cons ,item ,list))))))
```

In this example, the identifiers `set!` and `cons` are closed in the transformer environment, and thus will not be affected by the meanings of those identifiers in the usage environment `env`.

Some macros may be non-hygienic by design. For example, the following defines a `loop` macro that implicitly binds `exit` to an escape procedure. The binding of `exit` is intended to capture free references to `exit` in the body of the loop, so `exit` must be left free when the body is closed:

```
(define-syntax loop
  (transformer
   (lambda (exp env)
     (let ((body (cdr exp)))
       `(call-with-current-continuation
         (lambda (exit)
           (let f ()
             ,@(map (lambda (exp)
                      (make-syntactic-closure env '(exit)
                        exp))
                    body)
             (f))))))))
```

To assign meanings to the identifiers in a form, use `make-syntactic-closure` to close the form in a syntactic environment.

make-syntactic-closure *environment free-names form* procedure

Environment must be a syntactic environment, *free-names* must be a list of identifiers, and *form* must be a form. `make-syntactic-closure` constructs and returns a syntactic closure of *form* in *environment*, which can be used anywhere that *form* could have been used. All the identifiers used in *form*, except those explicitly excepted by *free-names*, obtain their meanings from *environment*.

Here is an example where *free-names* is something other than the empty list. It is instructive to compare the use of *free-names* in this example with its use in the `loop` example above: the examples are similar except for the source of the identifier being left free.

```
(define-syntax let1
  (transformer
   (lambda (exp env)
     (let ((id (cadr exp))
           (init (caddr exp))
           (exp (cadddr exp)))
       `((lambda (,id)
           ,(make-syntactic-closure env (list id) exp))
         ,(make-syntactic-closure env '() init))))))
```

`let1` is a simplified version of `let` that only binds a single identifier, and whose body consists of a single expression. When the body expression is syntactically closed in its original syntactic environment, the identifier that is to be bound by `let1` must be left free, so that it can be properly captured by the `lambda` in the output form.

To obtain a syntactic environment other than the usage environment, use `capture-syntactic-environment`.

capture-syntactic-environment *procedure* procedure

`capture-syntactic-environment` returns a form that will, when transformed, call *procedure* on the current syntactic environment. *Procedure* should compute and return a new form to be transformed, in that same syntactic environment, in place of the form.

An example will make this clear. Suppose we wanted to define a simple `loop-until` keyword equivalent to

```
(define-syntax loop-until
  (syntax-rules ()
    ((loop-until id init test return step)
     (letrec ((loop
               (lambda (id)
                 (if test return (loop step)))))
       (loop init)))))
```

The following attempt at defining `loop-until` has a subtle bug:

```
(define-syntax loop-until
  (transformer
   (lambda (exp env)
     (let ((id (cadr exp))
           (init (caddr exp))
           (test (cadddr exp))
           (return (cadddr (cdr exp)))
           (step (cadddr (cddr exp)))
           (close
            (lambda (exp free)
              (make-syntactic-closure env free exp))))
       `(letrec ((loop
                  (lambda (,id)
                    (if ,(close test (list id))
                        ,(close return (list id))
                        (loop ,(close step (list id)))))))
          (loop ,(close init '())))))))
```

This definition appears to take all of the proper precautions to prevent unintended captures. It carefully closes the subexpressions in their original syntactic environment and it leaves the `id` identifier free in the `test`, `return`, and `step` expressions, so that it will be captured by the binding introduced by the `lambda` expression. Unfortunately it uses the identifiers `if` and `loop` *within* that `lambda` expression, so if the user of `loop-until` just happens to use, say, `if` for the identifier, it will be inadvertently captured.

The syntactic environment that `if` and `loop` want to be exposed to is the one just outside the `lambda` expression: before the user's identifier is added to the syntactic environment, but after the identifier `loop` has been added. `capture-syntactic-environment` captures exactly that environment as follows:

```
(define-syntax loop-until
  (transformer
   (lambda (exp env)
     (let ((id (cadr exp))
           (init (caddr exp))
           (test (cadddr exp))
           (return (cadddr (cdr exp)))
           (step (cadddr (cddr exp)))
           (close
            (lambda (exp free)
              (make-syntactic-closure env free exp))))
       `(letrec ((loop
                  ,(capture-syntactic-environment
                    (lambda (env)
                      `(lambda (,id)
                         (,(make-syntactic-closure env '() `if)
                          ,(close test (list id))
                          ,(close return (list id))
                          (,(make-syntactic-closure env '() `loop)
                           ,(close step (list id)))))))))
          (loop ,(close init '())))))))
```

In this case, having captured the desired syntactic environment, it is convenient to construct syntactic closures of the identifiers `if` and the `loop` and use them in the body of the `lambda`.

A common use of `capture-syntactic-environment` is to get the transformer environment of a macro transformer:

```
(transformer
 (lambda (exp env)
   (capture-syntactic-environment
    (lambda (transformer-env)
      ...))))
```

Identifiers

This section describes the procedures that create and manipulate identifiers. Previous syntactic closure proposals did not have an identifier data type—they just used symbols. The identifier data type extends the syntactic closures facility to be compatible with the high-level `syntax-rules` facility.

As discussed earlier, an identifier is either a symbol or an *alias*. An alias is implemented as a syntactic closure whose *form* is an identifier:

```
(make-syntactic-closure env '() 'a) ⇒ an alias
```

Aliases are implemented as syntactic closures because they behave just like syntactic closures most of the time. The difference is that an alias may be bound to a new value (for example by `lambda` or `let-syntax`); other syntactic closures may not be used this way. If an alias is bound, then within the scope of that binding it is looked up in the syntactic environment just like any other identifier.

Aliases are used in the implementation of the high-level facility `syntax-rules`. A macro transformer created by `syntax-rules` uses a template to generate its output form, substituting subforms of the input form into the template. In a syntactic closures implementation, all of the symbols in the template are replaced by aliases closed in the transformer environment, while the output form itself is closed in the usage environment. This guarantees that the macro transformation is hygienic, without requiring the transformer to know the syntactic roles of the substituted input subforms.

identifier? *object* procedure

Returns `#t` if *object* is an identifier, otherwise returns `#f`. Examples:

```
(identifier? 'a)         ⇒ #t
(identifier? (make-syntactic-closure env '() 'a))
                         ⇒ #t

(identifier? "a")        ⇒ #f
(identifier? #\a)        ⇒ #f
(identifier? 97)         ⇒ #f
(identifier? #f)         ⇒ #f
(identifier? '(a))       ⇒ #f
(identifier? '#(a))      ⇒ #f
```

The predicate `eq?` is used to determine if two identifers are "the same". Thus `eq?` can be used to compare identifiers exactly as it would be used to compare symbols. Often, though, it is useful to know whether two identifiers "mean the same thing". For example, the `cond` macro uses the symbol `else` to identify the final clause in the conditional. A macro transformer for `cond` cannot just look for the symbol `else`, because the `cond` form might be the output of another macro transformer that replaced the symbol `else` with an alias. Instead the transformer must look for an identifier that "means the same thing" in the usage environment as the symbol `else` means in the transformer environment.

identifier=? *environment1 identifier1 environment2 identifier2* procedure

Environment1 and *environment2* must be syntactic environments, and *identifier1* and *identifier2* must be identifiers. `identifier=?` returns `#t` if the meaning of *identifier1* in *environment1* is the same as that of *identifier2* in *environment2*, otherwise it returns `#f`. Examples:

```
(let-syntax
    ((foo
      (transformer
       (lambda (form env)
         (capture-syntactic-environment
          (lambda (transformer-env)
            (identifier=? transformer-env 'x env 'x)))))))
  (list (foo)
        (let ((x 3))
          (foo))))
                          ⇒  (#t #f)

(let-syntax ((bar foo))
  (let-syntax
      ((foo
        (transformer
         (lambda (form env)
           (capture-syntactic-environment
            (lambda (transformer-env)
              (identifier=? transformer-env 'foo
                            env (cadr form))))))))
    (list (foo foo)
          (foo bar))))
                          ⇒  (#f #t)
```

Acknowledgments

The syntactic closures facility was invented by Alan Bawden and Jonathan Rees. The use of aliases to implement `syntax-rules` was invented by Alan Bawden (who prefers to call them "synthetic names"). Much of this proposal is derived from an earlier proposal by Alan Bawden.