Contents

1 Modules and macros

CHICKEN supports standard R5RS `syntax-rules` macros and a low-level macro system based on *explicit renaming*.

1.1 Macro definitions

1.1.1 define-syntax

`\[syntax\] (define-syntax IDENTIFIER TRANSFORMER)`

Defines a macro named `IDENTIFIER` that will transform an expression with `IDENTIFIER` in operator position according to `TRANSFORMER`. The transformer expression must be a procedure with three arguments or a `syntax-rules` form. If `syntax-rules` is used, the usual R5RS semantics apply. If `TRANSFORMER` is a procedure, then it will be called on expansion with the complete s-expression of the macro invocation, a rename procedure that hygienically renames identifiers and a comparison procedure that compares (possibly renamed) identifiers.

`define-syntax` may be used to define local macros that are visible throughout the rest of the body in which the definition occurred, i.e.

```
(let ()
  ...
  (define-syntax foo ...)
  (define-syntax bar ...)
  ...)
```

is expanded into

```
(let ()
  ...
  (letrec-syntax ((foo ...) (bar ...))
    ...) )
```

`syntax-rules` partially supports SRFI-46 in allowing the ellipsis identifier to be user-defined by passing it as the first argument to the `syntax-rules` form.

1.1.2 define-compiled-syntax

`\[syntax\] (define-compiled-syntax IDENTIFIER TRANSFORMER)`

Equivalent to `define-syntax`, but when compiled, will also define the macro at runtime.

1.1.3 syntax

`\[syntax\] (syntax EXPRESSION)`

Similar to `quote` but retains syntactical context information for embedded identifiers.

1.2 Explicit renaming macros

The low-level macro facility that CHICKEN provides is called explicit renaming and allows writing hygienic or nonhygienic macros procedurally. When given a lambda-expression instead of a `syntax-rules` form, `define-syntax` evaluates the procedure in a distinct expansion environment (initially having access to the exported identifiers of the `scheme` module). The procedure takes an expression and two other arguments and returns a transformed expression.

For example, the transformation procedure for a `call` macro such that `(call proc arg ...)` expands into `(proc arg ...)` can be written as

```
(lambda (exp rename compare)
  (cdr exp))
```

Expressions are represented as lists in the traditional manner, except that identifiers are represented as special uninterned symbols.

The second argument to a transformation procedure is a renaming procedure that takes the representation of an identifier as its argument and returns the representation of a fresh identifier that occurs nowhere else in the program. For example, the transformation procedure for a simplified version of the `let` macro might be written as

```
(lambda (exp rename compare)
  (let ((vars (map car (cadr exp)))
        (inits (map cadr (cadr exp)))
        (body (cddr exp)))
    `((lambda ,vars ,@body)
      ,@inits)))
```

This would not be hygienic, however. A hygienic `let` macro must rename the identifier `lambda` to protect it from being captured by a local binding. The renaming effectively creates an fresh alias for `lambda`, one that cannot be captured by any subsequent binding:

```
(lambda (exp rename compare)
  (let ((vars (map car (cadr exp)))
        (inits (map cadr (cadr exp)))
        (body (cddr exp)))
    `((,(rename 'lambda) ,vars ,@body)
      ,@inits)))
```

The expression returned by the transformation procedure will be expanded in the syntactic environment obtained from the syntactic environment of the macro application by binding any fresh identifiers generated by the renaming procedure to the denotations of the original identifiers in the syntactic environment in which the macro was defined. This means that

a renamed identifier will denote the same thing as the original identifier unless the transformation procedure that renamed the identifier placed an occurrence of it in a binding position.

Identifiers obtained from any two calls to the renaming procedure with the same argument will necessarily be the same, but will denote the same syntactical binding. It is an error if the renaming procedure is called after the transformation procedure has returned.

The third argument to a transformation procedure is a comparison predicate that takes the representations of two identifiers as its arguments and returns true if and only if they denote the same thing in the syntactic environment that will be used to expand the transformed macro application. For example, the transformation procedure for a simplified version of the `cond` macro can be written as

```
(lambda (exp rename compare)
  (let ((clauses (cdr exp)))
    (if (null? clauses)
        `(,(rename 'quote) unspecified)
        (let* ((first (car clauses))
               (rest (cdr clauses))
               (test (car first)))
          (cond ((and (identifier? test)
                      (compare test (rename 'else)))
                 `(,(rename 'begin) ,@(cdr first)))
                (else `(,(rename 'if)
                        ,test
                        (,(rename 'begin) ,@(cdr first))
                        (cond ,@rest)))))))))
```

In this example the identifier `else` is renamed before being passed to the comparison predicate, so the comparison will be true if and only if the test expression is an identifier that denotes the same thing in the syntactic environment of the expression being transformed as `else` denotes in the syntactic environment in which the `cond` macro was defined. If `else` were not renamed before being passed to the comparison predicate, then it would match a local variable that happened to be named `else`, and the macro would not be hygienic.

Some macros are 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` is not renamed.

```
(define-syntax loop
   (lambda (x r c)
     (let ((body (cdr x)))
       `(,(r 'call-with-current-continuation)
         (,(r 'lambda) (exit)
          (,(r 'let) ,(r 'f) () ,@body (,(r 'f))))))))
```

Suppose a `while` macro is implemented using `loop`, with the intent that `exit` may be used to escape from the `while` loop. The `while` macro cannot be written as

```
(define-syntax while
  (syntax-rules ()
    ((while test body ...)
     (loop (if (not test) (exit \#f))
           body ...))))
```

because the reference to `exit` that is inserted by the `while` macro is intended to be captured by the binding of `exit` that will be inserted by the `loop` macro. In other words, this `while` macro is not hygienic. Like `loop`, it must be written using procedurally:

```
(define-syntax while
   (lambda (x r c)
     (let ((test (cadr x))
           (body (cddr x)))
       `(,(r 'loop)
         (,(r 'if) (,(r 'not) ,test) (exit \#f))
         ,@body))))
```

1.3 Modules

To allow some control over visible bindings and to organize code at the global level, a simple module system is available. A *module* defines a set of toplevel expressions that are initially evaluated in an empty syntactical

environment. By *importing* other modules, exported value- and macro-bindings are made visible inside the environment of the module that imports them.

Note that modules are purely syntactical - they do not change the control flow or delay the execution of the contained toplevel forms. The body of a module is executed at load-time, when code is loaded or accessed via the `uses` declaration, just like normal toplevel expressions. Exported macro-definitions are compiled as well, and can be accessed in interpreted or compiled code by loading and importing the compiled file that contains the module.

A module is initially empty (has no visible bindings). You must at least import the `scheme` module to do anything useful.

1.3.1 module

```
\[syntax\] (module NAME (EXPORT ...) BODY ...)
```

Defines a module with the name `NAME`, a set of exported bindings and a contained sequence of toplevel expressions that are evaluated in an empty syntactical environment. `EXPORT` may be a symbol or a list of the form `(IDENTIFIER1 IDENTIFIER2 ...)`. In the former case the identifier given is exported from the module and can be imported at the toplevel or in other modules. The latter case exports all identifiers listed (this is a hint to the module expander to export bindings referenced by syntax-definitions which make use of them, but which would normally be internal to the module - that allows some optimization, which is currently not implemented but may be in the future).

Nested modules or modules not at toplevel (i.e. local modules) are not supported.

When compiled, the module information, including exported macros is stored in the generated binary and available when loading it into interpreted or compiled code. Note that this is different to normal macros (outside of module declarations), which are normally not exported from compiled code.

1.3.2 import

```
\[syntax\] (import IMPORT ...)
```

Imports module bindings into the currentl syntactical environment. The visibility of any imported bindings is limited to the current module, if used inside a module-definition, or to the current compilation unit, if compiled and used outside of a module.

Importing a module does not load or link it - this is a separate operation from importing its bindings.

`IMPORT` may be a module name, or an *import specifier*. An `IMPORT` defines a set of bindings that are to be made visible in the current scope.

only

```
\[import specifier\] (only IMPORT IDENTIFIER ...)
```

Only import the listed value- or syntax bindings from the set given by `IMPORT`.

except

```
\[import specifier\] (except IMPORT IDENTIFIER ...)
```

Remove the listed identifiers from the import-set defined by `IMPORT`.

rename

```
\[import specifier\] (rename IMPORT (OLD1 NEW1) ...)
```

Renames identifiers imported from `IMPORT`.

prefix

```
\[import specifier\] (prefix IMPORT SYMBOL)
```

Prefixes all imported identifiers with `SYMBOL`.

1.3.3 import-for-syntax

```
\[syntax\] (import-for-syntax IMPORT ...)
```

Similar to `import`, but imports exported bindings of a module into the environment in which macro transformers are evaluated.

Note: currently this isn't fully correct - value bindings are still imported into the normal environment because a separate import environment for syntax has not been implemented (syntactic bindings are kept separate correctly).

1.4 import libraries

import libraries allow the syntactical (compile-time) and run-time parts of a compiled module to be separated into a normal compiled file and a shared library that only contains macro definitions and module information. This reduces the size of executables and simplifies compiling code that uses modules for a different architecture than the machine the compiler is executing on (i.e. cross compilation).

By using the `emit-import-library` compiler-option or declaration, a separate file is generated that only contains syntactical information (including macros) for a module. `import` will automatically find and load an import library for a currently unknown module, if the import- library is either in the extension repository or the current include path. Import libraries may also be explicitly loaded into the compiler by using the `-extend` compiler option. Interpreted code can simply load the import library to make the module-definition available.

1.5 Predefined modules

Import libraries for the following modules are initially available:

`\[module\] scheme`

Exports the standard R5RS bindings.

`\[module\] chicken`

Everything from the `library`, `eval` and `expand` library units.

```
\[module\] extras
\[module\] data-structures
\[module\] lolevel
\[module\] posix
\[module\] regex
\[module\] srfi-1
\[module\] srfi-4
\[module\] srfi-13
\[module\] srfi-14
\[module\] srfi-18
```

```
\[module\] srfi-69
\[module\] tcp
\[module\] utils
```

Modules exporting the bindings from the respective library units.

```
\[module\] foreign
```

Exports all macros and procedures that are used to access foreign C/C++ code.

1.6 Examples of using modules

Here is a silly little test module to demonstrate how modules are defined and used:

; hello.scm

(module test (hello greet)

```
 (import scheme)

 (define-syntax greet
   (syntax-rules ()
     ((\_ whom)
      (begin
```

(display Hello,) (display whom) (display !n)))))

```
 (define (hello)
   (greet world) )  )
```

The module `test` exports one value (`hello`) and one syntax binding (`greet`). To use it in `csi`, the interpreter, simply load and import it:

```
 \#;1> ,l hello.scm
 ; loading hello.scm ...
 ; loading /usr/local/lib/chicken/4/scheme.import.so ...
 \#;1> (import test)
 \#;2> (hello)
 Hello, world !
 \#;3> (greet you)
 Hello, you !
```

The module can easily be compiled

```
% csc -s hello.scm
```

and used in an indentical manner:

```
\#;1> ,l hello.so
; loading hello.so ...
\#;1> (import test)
\#;2> (hello)
Hello, world !
\#;3> (greet you)
Hello, you !
```

If you want to keep macro-definitions in a separate file, use import libraries:

```
% csc -s hello.scm -j test
% csc -s test.import.scm

\#;1> ,l hello.so
; loading hello.so ...
\#;1> (import test)
; loading ./test.import.so ...
\#;2> (hello)
Hello, world !
\#;3> (greet you)
Hello, you !
```

If an import library (compiled or in source-form) is located somewhere in the extensions-repository or include path, it is automatically loaded on import. Otherwise you have to load it manually:

```
\#;1> ,l hello.so
; loading hello.so ...
\#;1> ,l test.import.so
; loading test.import.so ...
\#;1> (import test)
\#;2>
```

1.7 Caveats

Macros are currently not referentially transparent on the module level: internal Identifiers used by exported macros must be exported too or the macro expansion will contain references to unbound identifiers. This is a bug.

The macro- and module system has been implemented relatively recently and is likely to contain bugs. Please contact the maintainers if you encounter behaviour that you think is not correct or that triggers an error where there shouldn't be one.

Currently value bindings imported by `import` and `import-for-syntax` share the same import-environment.

Previous: Non-standard macros and special forms

Next: Declarations