Unsyntax

for version 0.0.3.24-c3fc, 8 April 2021

Marc Nieper-Wißkirchen (marc@unsyntax.org)

This manual is for Unsyntax (version 0.0.3.24-c3fc, 8 April 2021), an R^7RS implementation with extensions.

Table of Contents

1 Overview

Unsyntax is an implementation of the Scheme programming language, specifically of its R^{7}RS standard, and includes a number of extensions. Scheme itself is a dialect of the Lisp family of programming languages, amongst whose distinguishing features are its minimalism, lexical scope, proper tail recursion, first-class continuations, a unified namespace and hygienic macros. Unsyntax's license allows its use in the Scheme Requests for Implementation (SRFI).

Unsyntax evaluates Scheme expressions and compiles and runs Scheme programs by first expanding them into a minimal dialect of R^{7}RS (small) without any syntactic extensions. The resulting expression or program is then evaluated by an existing Scheme implementation.

As said, Unsyntax implements the R^{7}RS standard with a number of extensions. Besides implementing a number of SRFIs that do not possess a portable implementation otherwise, Unsyntax notably fully implements the syntax-case system of R^{6}RS and the R^{6}RS record system besides the R^{7}RS record system. It can be used as a test-bed for further additions to R^{7}RS, which may become part of the future R^{7}RS (large) standard.

Unsyntax is implemented in Scheme itself. To bootstrap Unsyntax, a fairly standard-conforming implementation of R^{7}RS (small) is needed. Currently, the build system presupposes the installation Alex Shinn's Chibi-Scheme implementation; see the Chibi-Scheme development page at `https://github.com/ashinn/chibi-scheme`.

On the target system, an implementation of a compatible backend is required. Currently, this is Chibi-Scheme as well. It should be easy to modify Unsyntax for other backends, like Chez Scheme or Gambit. Patches are very welcome.

Unsyntax was written by Marc Nieper-Wißkirchen.

2 Examples

Here are some examples of using Unsyntax:

This is an example session of Unsyntax's REPL:

```
$ unsyntax-scheme
Unsyntax 0.0.3
Copyright (C) 2020 Marc Nieper-Wißkirchen

This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.

#,> (+ 3 4)
7
#,> (import (srfi 1))
#,> (iota 3)
(0 1 2)
#,> ^C
```

This is an example session of running a program with Unsyntax's interpreter:

```
$ cat hello-world.scm
(import (scheme base)
        (scheme write))
(display "Hello, World!")
(newline)
$ unsyntax-scheme hello-world.scm
Hello, World!
```

This is an example session of compiling a program with Unsyntax and running it:

```
$ compile-unsyntax hello-world.scm
$ ./hello-world
Hello, World!
```

3 The Unsyntax programming language

3.1 The Base language

Unsyntax implements the Scheme programming language as described in the R^{7}RS standard.

3.2 Standard libraries

It implements the following standard libraries:

`(scheme base)`
: The `(scheme base)` library exports many of the procedures and syntax bindings that are traditionally associated with Scheme.

`(scheme case-lambda)`
: The `(scheme case-lambda)` library exports the `case-lambda` syntax.

`(scheme char)`
: The `(scheme char)` library provides the procedures for dealing with characters that involve potentially large tables when supporting all of Unicode.

`(scheme complex)`
: The `(scheme complex)` library exports procedures which are typically only useful with non-real numbers.

`(scheme cxr)`
: The `(scheme cxr)` library exports twenty-four procedures which are the compositions of from three to four `car` and `cdr` operations.

`(scheme eval)`
: The `(scheme eval)` library exports procedures for evaluating Scheme data as programs.

`(scheme file)`
: The `(scheme file)` library exports procedures for accessing files.

`(scheme inexact)`
: The `(scheme inexact)` library exports procedures which are typically only useful with inexact values.

`(scheme lazy)`
: The `(scheme lazy)` library exports procedures and syntax keywords for lazy evaluation.

`(scheme load)`
: The `(scheme load)` library exports procedures for loading Scheme expressions from files.

`(scheme process-context)`
: The `(scheme process-context)` library exports procedures for accessing the program's calling context.

`(scheme r5rs)`
: The `(scheme r5rs)` library provides the identifiers defined by R5RS, except that `transcript-on` and `transcript-off` are not present.

`(scheme read)`
: The `(scheme read)` library provides a procedure for reading Scheme objects.

`(scheme repl)`
: The `(scheme repl)` library exports the `interaction-environment` procedure.

`(scheme time)`
: The `(scheme time)` library provides access to time-related values.

`(scheme write)`
: The `(scheme write)` library provides procedures for writing Scheme objects.

3.3 R^{6}RS libraries

A few R^{6}RS libraries are also provided:

`(rnrs records syntactic (6))`
: Syntactic layer for R^{6}RS records.

`(rnrs records syntactic (6))`
: Procedural layer for R^{6}RS records.

`(rnrs records inspection (6))`
: Inspection for R^{6}RS records.

3.4 SRFI libraries

Additionally, it provides the following SRFI libraries:

`(srfi 1)`
`(scheme list)`
: List library.

`(srfi 2)` `And-let*`: an `and` with local bindings, a guarded `let*` special form.

`(srfi 8)` `Receive`: binding to multiple values.

`(srfi 28)` Basic format strings.

`(srfi 37)` `Args-fold`: a program argument processor.

`(srfi 59)` Vicinity.

`(srfi 64)` A Scheme API for test suites.

`(srfi 111)`
`(scheme box)`
: Boxes.

`(srfi 125)`
`(scheme hash-table)`
: Intermediate hash tables.

`(srfi 128)`
`(scheme comparator)`
: Comparators (reduced).

`(srfi 139)`
: Syntax parameters.

`(srfi 158)`
: Generators and accumulators.

`(srfi 188)`
: Splicing binding constructs for syntactic keywords.

`(srfi 190)`
: Coroutine generators.

`(srfi 206)`
: Auxiliary Syntax Keywords.

`(srfi 211 define-macro)`
: Old-style Lisp macros.

`(srfi 211 explicit-renaming)`
: Explicitly renaming macros.

`(srfi 211 identifier-syntax)`
: Identifier syntax.

`(srfi 211 ir-macro-transformer)`
: Implicitly renaming macros.

`(srfi 211 r4rs)`
: The low-level macro facility of the R^{4}RS.

`(srfi 211 syntactic-closures)`
: The syntactic-closures macro facility.

`(srfi 211 syntax-case)`
: `Syntax-case`

`(srfi 211 syntax-parameter)`
: Syntax parameters.

`(srfi 211 variable-transformer)`
: Variable transformers.

`(srfi 211 with-ellipsis)`
: Custom ellipsis identifiers for `syntax-case` macros.

`(srfi 212)`
: Aliases.

`(srfi 213)`
: Identifier properties.

The `compile-unsyntax` command implements SRFI 138.

3.5 Extensions

The `(unsyntax)` library exports a number of useful bindings, including not yet standardized variables and keywords.

`meta` . *form* [Special Form]

The `meta` keyword is a prefix that turns any form in a *meta definition*, which can be used where other definitions are allowed. The *form* is evaluated at expand-time and defined values become available at expand-time in the meta definition itself and in subsequent forms.

Meta definitions are useful for defining expand-time helper procedures and other variables for use by transformer expressions in the same module as in the following example:

```
(meta define (construct-name key . args)
  (datum->syntax key
                 (string->symbol
                  (apply string-append
                         (map (lambda (x)
                                (cond
                                 ((string? x) x)
                                 ((identifier? x)
                                 (symbol->string (syntax->datum x)))))
                              args)))))

(define-syntax define-zoo
  (lambda (stx)
    (syntax-case stx ()
      ((k prefix)
       #`(begin
           (define #,(construct-name #'k #'prefix "-lion") 'lion)
           (define #,(construct-name #'k #'prefix "-tiger") 'tiger))))))

(define-zoo local)
local-lion ⇒ lion
local-tiger ⇒ tiger
```

`import` *import-spec...* [Special Form]

An `import` form can appear wherever a definition is allowed. It lexically imports the bindings as specified by the *import specs* into the enclosing definition context.

`import-only` *import-spec...* [Special Form]

An `import-only` form can appear wherever a definition is allowed. It lexically hides all existing bindings and then imports the bindings as specified by the *import specs* into the enclosing lexical context.

`module` [*module name*] (*export-spec...*) *body...* [Special Form]

A `module` form can appear wherever a definition is allowed. When a `module` form is expanded, the body is expanded but the created bindings are not visible outside. If *module name* is given, it is bound to an expand-time representation of the module,

which can be used in place of a library reference in local `import` and `import-only` forms. The exported bindings are determined according to the *export-specs*. If *module name* is omitted, an anonymous module is created and implicitly imported through `import` where it is defined.

3.6 Implementation notes

In implementing the R^{7}RS standard, a few choices had to be made:

- The R^{7}RS grammar for a program is ambigious when there is more than one import declaration at the beginning as it is unclear whether any subsequent form is already an expression or definition. The use of more than one import declaration is therefore deprecated because one import declaration is enough as an import declaration can take more than import set.

  For legacy code, the ambiguity is resolved as follows: The line between the import declarations and the body of a program is drawn either before at the first form that is not a pair whose car is the identifier `import` or whenever the identifier `import` is imported by the previous import declaration, whatever comes earlier.
- Inclusion expressions in Unsyntax search for files relative to the directory, which contains the including file. The syntactic context of the included expression is the syntactic context of the corresponding `include` or `include-ci` keyword.
- The clauses of a `cond-expand` form are matched unhygienically, that is by symbol and not by identifier equality.
- Field names in record-type definitions are identifiers and not symbols and are thus matched hygienically, that is by identifier and not by symbol equality.
- A program is evaluated by first expanding it and then executing it. During the expansion phase, macro transformers are applied. In case of ahead-of-time compilation, the expansion phase may be separated from the final execution. In no case will a library or a program being expanded more than once during the evaluation of a program. During expansion of a program, no library is executed more than once. The same is true for the execution of a program. In case expansion and execution of a program is not separated, every library is executed at most once. Only those libraries are evaluated, whose bindings are referenced, directly or indirectly, in the program.
- In evaluating bodies, libraries, and programs, Unsyntax follows the R^{6}RS semantics. In particular, for the definition of a variable, expansion is deferred until after all of the definitions have been seen. A use of a syntax form can therefore appear before its definition.
- The clauses of a `cond-expand` form allow both expressions and definitions. In particular, like `begin`, `cond-expand` doesn't have to be an expression.

4 Invoking `unsyntax-scheme`

The format for running the `unsyntax-scheme` program to start the REPL is:

```
unsyntax-scheme option ...
```

The format for running the `unsyntax-scheme` program to run a Scheme program is:

```
unsyntax-scheme option ... -- program arg ...
```

`unsyntax-scheme` supports the following options:

`-A` *directory*
: Append *directory* to the library search paths.

`-D` *feature*
: Add *feature* to the list of feature identifiers.

`-I` *directory*
: Prepend *directory* to the library search paths.

`--help`
`-h`
: Print an informative help message on standard output and exit successfully.

`--version`
`-v`
: Print the version number and licensing information of Unsyntax on standard output and then exit successfully.

If more than one option is given, they are processed in left-to-right order.

5 Invoking `compile-unsyntax`

The format for running the `compile-unsyntax` program is:

```
compile-unsyntax option ... file
```

`compile-unsyntax` compiles the Scheme program *file* into an executable, whose filename is the one of *file* stripped from the `.scm` extension if it has one or `a.out` if not.

`unsyntax-scheme` supports the following options:

`-A` *directory*
: Append *directory* to the library search paths.

`-D` *feature*
: Add *feature* to the list of feature identifiers.

`-I` *directory*
: Prepend *directory* to the library search paths.

`-h`
: Print an informative help message on standard output and exit successfully.

`-o` *file*
: Place the output into *file*.

`-v`
: Print the version number and licensing information of Unsyntax on standard output and then exit successfully.

If more than one option is given, they are processed in left-to-right order.

6 Invoking `expand-unsyntax`

The format for running the `expand-unsyntax` program is:

```
expand-unsyntax option ... file
```

`expand-unsyntax` expand the Scheme program *file* into a minimal dialect of R^{7}RS.

`unsyntax-scheme` supports the following options:

`-A` *directory*
: Append *directory* to the library search paths.

`-D` *feature*
: Add *feature* to the list of feature identifiers.

`-I` *directory*
: Prepend *directory* to the library search paths.

`--help`
`-h`
: Print an informative help message on standard output and exit successfully.

`-o` *file*
: Place the output into *file* instead of using stdout.

`--version`
`-v`
: Print the version number and licensing information of Unsyntax on standard output and then exit successfully.

If more than one option is given, they are processed in left-to-right order.

7 Environment variable

The environment variable `UNSYNTAX_LIBRARY_PATH` can hold a colon-separated list of library search paths to be used instead of the usual system-defined path.

8 Reporting bugs

To report bugs, suggest enhancements or otherwise discuss Unsyntax, please send electronic mail to `bug@unsyntax.org`. You can also join the development at `https://gitlab.com/nieper/unsyntax`.

For bug reports, please include enough information for the maintainers to reproduce the problem. Generally speaking, that means:

- The version numbers of Unsyntax (which you can find by running '`unsyntax-scheme --version`') and any other program(s) or manual(s) involved.
- Hardware and operating system names and versions.
- The contents of any input files necessary to reproduce the bug.
- The expected behavior and/or output.
- A description of the problem and samples of any erroneous output.
- Options you gave to `configure` other than specifying installation directories.
- Anything else that you think would be helpful.

When in doubt whether something is needed or not, include it. It's better to include too much than to leave out something important.

Patches are welcome; if possible, please make them with '`diff -c`' (see Section "Overview" in *Comparing and Merging Files*) and include `ChangeLog` entries (see Section "Change Log" in *The GNU Emacs Manual*). Please follow the existing coding style. You can also issue merge requests at `https://gitlab.com/nieper/unsyntax`.

Concept index