Syntax: Meta-Programming Helpers

Version 4.1.1

October 5, 2008

Contents

1 Syntax Object Helpers

1.1 Deconstructing Syntax Objects

```
(require syntax/stx)
```

```
(stx-null? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is either the empty list or a syntax object representing the empty list (i.e., `syntax-e` on the syntax object returns the empty list).

```
(stx-pair? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is either a pair or a syntax object representing a pair (see syntax pair).

```
(stx-list? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is a list, or if it is a sequence of pairs leading to a syntax object such that `syntax->list` would produce a list.

```
(stx->list stx-list) → list?
  stx-list : stx-list?
```

Produces a list by flatting out a trailing syntax object using `syntax->list`.

```
(stx-car v) → any
  v : stx-pair?
```

Takes the car of a syntax pair.

```
(stx-cdr v) → any
  v : stx-pair?
```

Takes the cdr of a syntax pair.

```
(module-or-top-identifier=? a-id b-id) → boolean?
```

a-id : identifier?
b-id : identifier?

Returns #t if *a-id* and *b-id* are free-identifier=?, or if *a-id* and *b-id* have the same name (as extracted by syntax-e) and *a-id* has no binding other than at the top level.

This procedure is useful in conjunction with syntax-case* to match procedure names that are normally bound by MzScheme. For example, the include macro uses this procedure to recognize build-path; using free-identifier=? would not work well outside of module, since the top-level build-path is a distinct variable from the MzScheme export (though it's bound to the same procedure, initially).

1.2 Matching Fully-Expanded Expressions

```
(require syntax/kerncase)
```

```
(kernel-syntax-case stx-expr trans?-expr clause ...)
```

A syntactic form like syntax-case*, except that the literals are built-in as the names of the primitive PLT Scheme forms as exported by scheme/base; see §1.2.3.1 "Fully Expanded Programs".

The *trans?-expr* boolean expression replaces the comparison procedure, and instead selects simply between normal-phase comparisons or transformer-phase comparisons. The *clause*s are the same as in syntax-case*.

The primitive syntactic forms must have their normal bindings in the context of the kernel-syntax-case expression. Beware that kernel-syntax-case does not work in a module whose language is mzscheme, since the binding of if from mzscheme is different than the primitive if.

```
(kernel-syntax-case* stx-expr trans?-expr (extras ...) clause ...)
```

A syntactic form like kernel-syntax-case, except that it takes an additional list of extra literals that are in addition to the primitive PLT Scheme forms.

```
(kernel-form-identifier-list) → (listof indentifier?)
```

Returns a list of identifiers that are bound normally, for-syntax, and for-template to the primitive PLT Scheme forms for expressions. This function is useful for generating a list of stopping points to provide to local-expand.

1.3 Hashing on bound-identifier=? and free-identifier=?

```
(require syntax/boundmap)
```

```
(make-bound-identifier-mapping) → bound-identifier-mapping?
```

Produces a hash-table-like value for storing a mapping from syntax identifiers to arbitrary values.

The mapping uses `bound-identifier=?` to compare mapping keys, but also uses a hash table based on symbol equality to make the mapping efficient in the common case (i.e., where non-equivalent identifiers are derived from different symbolic names).

```
(bound-identifier-mapping? v) → boolean?
  v : any/c
```

Returns `#t` if *v* was produced by `make-bound-identifier-mapping`, `#f` otherwise.

```
(bound-identifier-mapping-get  bound-map
                               id
                              [failure-thunk]) → any
  bound-map : bound-identifier-mapping?
  id : identifier?
  failure-thunk : any/c
                = (lambda () (raise (make-exn:fail ....)))
```

Like `hash-table-get` for bound-identifier mappings.

```
(bound-identifier-mapping-put! bound-map
                               id
                               v)         → void?
  bound-map : bound-identifier-mapping?
  id : identifier?
  v : any/c
```

Like `hash-table-put!` for bound-identifier mappings.

```
(bound-identifier-mapping-for-each bound-map
                                   proc)      → void?
  bound-map : boud-identifier-mapping?
  proc : (identifier? any/c . -> . any)
```

Like `hash-table-for-each`.

```
(bound-identifier-mapping-map bound-map
                              proc)     → (listof any?)
  bound-map : bound-identifier-mapping?
  proc : (identifier? any/c . -> . any)
```

Like `hash-table-map`.

```
(make-free-identifier-mapping) → free-identifier-mapping?
```

Produces a hash-table-like value for storing a mapping from syntax identifiers to arbitrary values.

The mapping uses `free-identifier=?` to compare mapping keys, but also uses a hash table based on symbol equality to make the mapping efficient in the common case (i.e., where non-equivalent identifiers are derived from different symbolic names at their definition sites).

```
(free-identifier-mapping? v) → boolean?
  v : any/c
```

Returns `#t` if *v* was produced by `make-free-identifier-mapping`, `#f` otherwise.

```
(free-identifier-mapping-get free-map
                             id
                            [failure-thunk]) → any
  free-map : free-identifier-mapping?
  id : identifier?
  failure-thunk : any/c
                = (lambda () (raise (make-exn:fail ....)))
```

Like `hash-table-get` for free-identifier mappings.

```
(free-identifier-mapping-put! free-map id v) → void?
  free-map : free-identifier-mapping?
  id : identifier?
  v : any/c
```

Like `hash-table-put!` for free-identifier mappings.

```
(free-identifier-mapping-for-each free-map
                                  proc)     → void?
```

```
  free-map : free-identifier-mapping?
  proc : (identifier? any/c . -> . any)
```

Like `hash-table-for-each`.

```
(free-identifier-mapping-map free-map proc) → (listof any?)
  free-map : free-identifier-mapping?
  proc : (identifier? any/c . -> . any)
```

Like `hash-table-map`.

```
(make-module-identifier-mapping) → module-identifier-mapping?
(module-identifier-mapping? v) → boolean?
  v : any/c
(module-identifier-mapping-get  module-map
                                id
                               [failure-thunk]) → any
  module-map : module-identifier-mapping?
  id : identifier?
  failure-thunk :  any/c
                = (lambda () (raise (make-exn:fail ....)))
(module-identifier-mapping-put! module-map
                                id
                                v)          → void?
  module-map : module-identifier-mapping?
  id : identifier?
  v : any/c
(module-identifier-mapping-for-each module-map
                                    proc)       → void?
  module-map : module-identifier-mapping?
  proc : (identifier? any/c . -> . any)
(module-identifier-mapping-map module-map
                               proc)       → (listof any?)
  module-map : module-identifier-mapping?
  proc : (identifier? any/c . -> . any)
```

The same as `make-module-identifier-mapping`, etc.

1.4 Rendering Syntax Objects with Formatting

```
(require syntax/to-string)
```

```
(syntax->string stx-list) → string?
  stx-list : stx-list?
```

Builds a string with newlines and indenting according to the source locations in *stx-list*; the outer pair of parens are not rendered from *stx-list*.

1.5 Computing the Free Variables of an Expression

```
(require syntax/free-vars)
```

```
(free-vars expr-stx) → (listof identifier?)
  expr-stx : syntax?
```

Returns a list of free `lambda`- and `let`-bound identifiers in *expr-stx*. The expression must be fully expanded (§1.2.3.1 "Fully Expanded Programs").

1.6 Legacy Zodiac Interface

```
(require syntax/zodiac)
(require syntax/zodiac-unit)
(require syntax/zodiac-sig)
```

The interface is similar to Zodiac—enough to be useful for porting—but different in many ways. See the source `"zodiac-sig.ss"` for details. New software should not use this compatibility layer.

2 Module-Processing Helpers

2.1 Reading Module Source Code

```
(require syntax/modread)
```

```
(with-module-reading-parameterization thunk) → any
  thunk : (-> any)
```

Calls *thunk* with all reader parameters reset to their default values.

```
(check-module-form stx
                   expected-module-sym
                   source-v)
 → (or/c syntax? false/c)
  stx : (or/c syntax? eof-object?)
  expected-module-sym : symbol?
  source-v : (or/c string? false/c)
```

Inspects `stx` to check whether evaluating it will declare a module named `expected-module-sym`—at least if `module` is bound in the top-level to MzScheme's `module`. The syntax object `stx` can contain a compiled expression. Also, `stx` can be an end-of-file, on the grounds that `read-syntax` can produce an end-of-file.

If `stx` can declare a module in an appropriate top-level, then the `check-module-form` procedure returns a syntax object that certainly will declare a module (adding explicit context to the leading `module` if necessary) in any top-level. Otherwise, if `source-v` is not `#f`, a suitable exception is raised using the `write` form of the source in the message; if `source-v` is `#f`, `#f` is returned.

If `stx` is eof or eof wrapped as a syntax object, then an error is raised or `#f` is returned.

2.2 Getting Module Compiled Code

```
(require syntax/modcode)
```

```
(get-module-code  module-path-v
                 [compiled-subdir
                  compile-proc
                  ext-proc
                  #:choose choose-proc
                  #:notify notify-proc
                  #:src-reader read-syntax-proc]) → any
  module-path-v : module-path?
  compiled-subdir : (and/c path-string? relative-path?)
                  = "compiled"
  compile-proc : (any/c . -> . any) = compile
  ext-proc : (or/c false/c (path? boolean? . -> . any)) = #f
  choose-proc : (path? path? path?
                 . -> .
                 (or/c (symbols 'src 'zo 'so) false/c))
              = (lambda (src zo so) #f)
  notify-proc : (any/c . -> . any) = void
  read-syntax-proc : (any/c input-port? . -> . syntax?)
                   = read-syntax
```

Returns a compiled expression for the declaration of the module specified by *module-path-v*.

The *compiled-subdir* argument defaults to "compiled"; it specifies the sub-directory to search for a compiled version of the module.

The *compile-proc* argument defaults to compile. This procedure is used to compile module source if an already-compiled version is not available.

The *ext-proc* argument defaults to #f. If it is not #f, it must be a procedure of two arguments that is called when a native-code version of path is should be used. In that case, the arguments to *ext-proc* are the path for the extension, and a boolean indicating whether the extension is a _loader file (#t) or not (#f).

The *choose-proc* argument is a procedure that takes three paths: a source path, a ".zo" file path, and an extension path (for a non-_loader extension). Some of the paths may not exist. The result should be either 'src, 'zo, 'so, or #f, indicating which variant should be used or (in the case of #f) that the default choice should be used.

The default choice is computed as follows: if a ".zo" version of path is available and newer than path itself (in one of the directories specified by *compiled-subdir*), then it is used instead of the source. Native-code versions of path are ignored, unless only a native-code non-_loader version exists (i.e., path itself does not exist). A _loader extension is selected a last resort.

If an extension is prefered or is the only file that exists, it is supplied to *ext-proc* when *ext-proc* is #f, or an exception is raised (to report that an extension file cannot be used) when *ext-proc* is `#f`.

If *notify-proc* is supplied, it is called for the file (source, `".zo"` or extension) that is chosen.

If *read-syntax-proc* is provided, it is used to read the module from a source file (but not from a bytecode file).

```
(moddep-current-open-input-file)
 → (path-string? . -> . input-port?)
(moddep-current-open-input-file proc) → void?
  proc : (path-string? . -> . input-port?)
```

A parameter whose value is used like `open-input-file` to read a module source or `".zo"` file.

```
(struct (exn:get-module-code exn) (path))
  path : path?
```

An exception structure type for exceptions raised by `get-module-code`.

2.3 Resolving Module Paths to File Paths

```
(require syntax/modresolve)
```

```
(resolve-module-path module-path-v
                     rel-to-path-v) → path?
  module-path-v : module-path?
  rel-to-path-v : (or/c path-string? (-> any) false/c)
```

Resolves a module path to filename path. The module path is resolved relative to *rel-to-path-v* if it is a path string (assumed to be for a file), to the directory result of calling the thunk if it is a thunk, or to the current directory otherwise.

```
(resolve-module-path-index module-path-index
                           rel-to-path-v)    → path?
  module-path-index : module-path-index?
  rel-to-path-v : (or/c path-string? (-> any) false/c)
```

Like `resolve-module-path` but the input is a module path index; in this case, the *rel-to-path-v* base is used where the module path index contains the “self” index. If *module-*

path-index depends on the "self" module path index, then an exception is raised unless *rel-to-path-v* is a path string.

2.4 Simplifying Module Paths

```
(require syntax/modcollapse)
```

```
(collapse-module-path module-path-v
                      rel-to-module-path-v)
 → (or/c path? module-path?)
  module-path-v : module-path?
  rel-to-module-path-v : any/c
```

Returns a "simplified" module path by combining *module-path-v* with *rel-to-module-path-v*, where the latter must have the form `'(lib ....)` or a symbol, `'(file <string>)`, `'(planet ....)`, a path, or a thunk to generate one of those.

The result can be a path if *module-path-v* contains a path element that is needed for the result, or if *rel-to-module-path-v* is a non-string path that is needed for the result; otherwise, the result is a module path in the sense of `module-path?`.

When the result is a `'lib` or `'planet` module path, it is normalized so that equivalent module paths are represented by `equal?` results.

```
(collapse-module-path-index module-path-index
                            rel-to-module-path-v)
 → (or/c path? module-path?)
  module-path-index : module-path-index?
  rel-to-module-path-v : any/c
```

Like `collapse-module-path`, but the input is a module path index; in this case, the *rel-to-module-path-v* base is used where the module path index contains the "self" index.

2.5 Inspecting Modules and Module Dependencies

```
(require syntax/moddep)
```

Re-exports `syntax/modread`, `syntax/modcode`, `syntax/modcollapse`, and `syntax/modresolve`, in addition to the following:

```
(show-import-tree module-path-v) → void?
```

module-path-v : `module-path?`

A debugging aid that prints the import hierarchy starting from a given module path.

3 Macro Transformer Helpers

3.1 Extracting Inferred Names

```
(require syntax/name)
```

```
(syntax-local-infer-name stx) → (or/c symbol? false/c)
  stx : syntax?
```

Similar to `syntax-local-name` except that *stx* is checked for an `'inferred-name` property (which overrides any inferred name). If neither `syntax-local-name` nor `'inferred-name` produce a name, then a name is constructed from the source-location information in *stx*, if any. If no name can be constructed, the result is `#f`.

3.2 Support for `local-expand`

```
(require syntax/context)
```

```
(build-expand-context v) → list?
  v : (or/c symbol? list?)
```

Returns a list suitable for use as a context argument to `local-expand` for an internal-definition context. The *v* argument represents the immediate context for expansion. The context list builds on `(syntax-local-context)` if it is a list.

```
(generate-expand-context) → list?
```

Calls `build-expand-context` with a generated symbol.

3.3 Parsing `define`-like Forms

```
(require syntax/define)
```

```
(normalize-definition  defn-stx
                       lambda-id-stx
                      [check-context?
                       opt+kws?])      → identifier? syntax?
  defn-stx : syntax?
```

```
lambda-id-stx : identifier?
check-context? : boolean? = #t
opt+kws? : boolean? = #t
```

Takes a definition form whose shape is like `define` (though possibly with a different name) and returns two values: the defined identifier and the right-hand side expression.

To generate the right-hand side, this function may need to insert uses of `lambda`. The *`lambda-id-stx`* argument provides a suitable `lambda` identifier.

If the definition is ill-formed, a syntax error is raised. If *`check-context?`* is true, then a syntax error is raised if `(syntax-local-context)` indicates that the current context is an expression context. The default value of *`check-context?`* is `#t`.

If `opt-kws?` is `#t`, then arguments of the form `[id expr]`, `keyword id`, and `keyword [id expr]` are allowed, and they are preserved in the expansion.

3.4 Expanding `define-struct`-like Forms

```
(require syntax/struct)
```

```
(parse-define-struct stx orig-stx) → identifier?
                                     (or/c identifier? false/c)
                                     (listof identifier?)
                                     syntax?
  stx : syntax?
  orig-stx : syntax?
```

Parses *`stx`* as a `define-struct` form, but uses *`orig-stx`* to report syntax errors (under the assumption that *`orig-stx`* is the same as *`stx`*, or that they at least share sub-forms). The result is four values: an identifier for the struct type name, a identifier or #f for the super-name, a list of identifiers for fields, and a syntax object for the inspector expression.

```
(build-struct-names  name-id
                     field-ids
                     omit-sel?
                     omit-set?
                    [src-stx])  → (listof identifier?)
  name-id : identifier?
  field-ids : (listof identifier?)
  omit-sel? : boolean?
  omit-set? : boolean?
  src-stx : (or/c syntax? false/c) = #f
```

Generates the names bound by `define-struct` given an identifier for the struct type name and a list of identifiers for the field names. The result is a list of identifiers:

- `struct:name-id`
- `make-name-id`
- `name-id?`
- `name-id-field`, for each `field` in `field-ids`.
- `set-name-id-field!` (getter and setter names alternate).
- ....

If `omit-sel?` is true, then the selector names are omitted from the result list. If `omit-set?` is true, then the setter names are omitted from the result list.

The default `src-stx` is `#f`; it is used to provide a source location to the generated identifiers.

```
(build-struct-generation  name-id
                          field-ids
                          omit-sel?
                          omit-set?
                         [super-type
                          prop-value-list
                          immutable-k-list])
 → (listof identifier?)
  name-id : identifier?
  field-ids : (listof identifier?)
  omit-sel? : boolean?
  omit-set? : boolean?
  super-type : any/c = #f
  prop-value-list : list? = empty
  immutable-k-list : list? = empty
```

Takes the same arguments as `build-struct-names` and generates an S-expression for code using `make-struct-type` to generate the structure type and return values for the identifiers created by `build-struct-names`. The optional `super-type`, `prop-value-list`, and `immutable-k-list` parameters take S-expression values that are used as the corresponding arguments to `make-struct-type`.

```
(build-struct-generation*  all-name-ids
                           name-id
                           field-ids
                           omit-sel?
                           omit-set?
                          [super-type
                           prop-value-list
                           immutable-k-list])
 → (listof identifier?)
  all-name-ids : (listof identifier?)
  name-id : identifier?
  field-ids : (listof identifier?)
  omit-sel? : boolean?
  omit-set? : boolean?
  super-type : any/c = #f
  prop-value-list : list? = empty
  immutable-k-list : list? = empty
```

Like `build-struct-generation`, but given the names produced by `build-struct-names`, instead of re-generating them.

```
(build-struct-expand-info name-id
                          field-ids
                          omit-sel?
                          omit-set?
                          base-name
                          base-getters
                          base-setters) → any
  name-id : identifier?
  field-ids : (listof identifier?)
  omit-sel? : boolean?
  omit-set? : boolean?
  base-name : (or/c identifier? boolean?)
  base-getters : (listof (or/c identifier? false/c))
  base-setters : (listof (or/c identifier? false/c))
```

Takes the same arguments as `build-struct-names`, plus a parent identifier/`#t`/`#f` and a list of accessor and mutator identifiers (possibly ending in `#f`) for a parent type, and generates an S-expression for expansion-time code to be used in the binding for the structure name. A `#t` for the *`base-name`* means no super-type, `#f` means that the super-type (if any) is unknown, and an identifier indicates the super-type identifier.

```
(struct-declaration-info? v) → boolean?
  v : any/c
```

Returns #t if x has the shape of expansion-time information for structure type declarations, #f otherwise. See §4.6 "Structure Type Transformer Binding".

```
(generate-struct-declaration  orig-stx
                              name-id
                              super-id-or-false
                              field-id-list
                              current-context
                              make-make-struct-type
                             [omit-sel?
                              omit-set?])            → syntax?
  orig-stx : syntax?
  name-id : identifier?
  super-id-or-false : (or/c identifier? false/c)
  field-id-list : (listof identifier?)
  current-context : any/c
  make-make-struct-type : procedure?
  omit-sel? : boolean? = #f
  omit-set? : boolean? = #f
```

This procedure implements the core of a define-struct expansion.

The generate-struct-declaration procedure is called by a macro expander to generate the expansion, where the *name-id*, *super-id-or-false*, and *field-id-list* arguments provide the main parameters. The *current-context* argument is normally the result of syntax-local-context. The *orig-stx* argument is used for syntax errors. The optional *omit-sel?* and *omit-set?* arguments default to #f; a #t value suppresses definitions of field selectors or mutators, respectively.

The make-struct-type procedure is called to generate the expression to actually create the struct type. Its arguments are *orig-stx*, name-id-stx, defined-name-stxes, and super-info. The first two are as provided originally to generate-struct-declaration, the third is the set of names generated by build-struct-names, and the last is super-struct info obtained by resolving *super-id-or-false* when it is not #f, #f otherwise.

The result should be an expression whose values are the same as the result of make-struct-type. Thus, the following is a basic *make-make-struct-type*:

```
(lambda (orig-stx name-stx defined-name-stxes super-info)
  #`(make-struct-type '#,name-stx
                       #,(and super-info (list-ref super-info 0))
                       #,(/ (- (length defined-name-stxes) 3) 2)
                       0 #f))
```

but an actual *make-make-struct-type* will likely do more.

3.5 Resolving include-like Paths

```
(require syntax/path-spec)
```

```
(resolve-path-spec path-spec-stx
                   source-stx
                   expr-stx
                   build-path-stx) → complete-path?
  path-spec-stx : syntax?
  source-stx : syntax?
  expr-stx : syntax?
  build-path-stx : syntax?
```

Resolves the syntactic path specification *path-spec-stx* as for `include`.

The *source-stx* specifies a syntax object whose source-location information determines relative-path resolution. The *expr-stx* is used for reporting syntax errors. The *build-path-stx* is usually `#'build-path`; it provides an identifier to compare to parts of *path-spec-stx* to recognize the `build-path` keyword.

4 Reader Helpers

4.1 Raising exn:fail:read

```
(require syntax/readerr)
```

```
(raise-read-error msg-string
                  source
                  line
                  col
                  pos
                  span)      → any
  msg-string : string?
  source : any/c
  line : (or/c number? false/c)
  col : (or/c number? false/c)
  pos : (or/c number? false/c)
  span : (or/c number? false/c)
```

Creates and raises an `exn:fail:read` exception, using *msg-string* as the base error message.

Source-location information is added to the error message using the last five arguments (if the `error-print-source-location` parameter is set to `#t`). The *source* argument is an arbitrary value naming the source location—usually a file path string. Each of the *line*, *pos* arguments is `#f` or a positive exact integer representing the location within `source-name` (as much as known), *col* is a non-negative exact integer for the source column (if known), and *span* is `#f` or a non-negative exact integer for an item range starting from the indicated position.

The usual location values should point at the beginning of whatever it is you were reading, and the span usually goes to the point the error was discovered.

```
(raise-read-eof-error msg-string
                      source
                      line
                      col
                      pos
                      span)      → any
  msg-string : string?
  source : any/c
  line : (or/c number? false/c)
  col : (or/c number? false/c)
```

```
pos : (or/c number? false/c)
span : (or/c number? false/c)
```

Like `raise-read-error`, but raises `exn:fail:read:eof` instead of `exn:fail:read`.

4.2 Module Reader

```
(require syntax/module-reader)
```

The `syntax/module-reader` language provides support for defining `#lang` readers. In its simplest form, the only thing that is needed in the body of a `syntax/module-reader` is the name of the module that will be used in the language position of read modules; using keywords, the resulting readers can be customized in a number of ways.

```
(#%module-begin module-path)
(#%module-begin module-path reader-option ... body ....)

reader-option = #:read         read-expr
              | #:read-syntax read-syntax-expr
              | #:wrapper1     wrapper1-expr
              | #:wrapper2     wrapper2-expr
              | #:whole-body-readers? whole?-expr
```

Causes a module written in the `syntax/module-reader` language to define and provide `read` and `read-syntax` functions, making the module an implementation of a reader. In particular, the exported reader functions read all S-expressions until an end-of-file, and package them into a new module in the *`module-path`* language.

That is, a module *`something`*`/lang/reader` implemented as

```
(module reader syntax/module-reader
  module-path)
```

creates a reader that converts `#lang` *`something`* into

```
(module name-id module-path
  ....)
```

where *`name-id`* is derived from the name of the port used by the reader.

For example, `scheme/base/lang/reader` is implemented as

```
(module reader syntax/module-reader
  scheme/base)
```

The reader functions can be customized in a number of ways, using keyword markers in the syntax of the reader module. A `#:read` and `#:read-syntax` keywords can be used to specify functions other than `read` and `read-syntax` to perform the reading. For example, you can implement a §"**Honu**" reader using:

```
(module reader syntax/module-reader
  honu
  #:read read-honu
  #:read-syntax read-honu-syntax)
```

You can also use the (optional) module body to provide more definitions that might be needed to implement your reader functions. For example, here is a case-insensitive reader for the `scheme/base` language:

```
(module insensitive syntax/module-reader
  scheme/base
  #:read (wrap read) #:read-syntax (wrap read-syntax)
  (define ((wrap reader) . args)
    (parameterize ([read-case-sensitive #f]) (apply reader args))))
```

In many cases, however, the standard `read` and `read-syntax` are fine, as long as you can customize the dynamic context they're invoked at. For this, `#:wrapper1` can specify a function that can control the dynamic context in which the reader functions are called. It should evaluate to a function that consumes a thunk and invokes it in the right context. Here is an alternative definition of the case-insensitive language using `#:wrapper1`:

```
(module insensitive syntax/module-reader
  scheme/base
  #:wrapper1 (lambda (t)
               (parameterize ([read-case-sensitive #f])
                 (t))))
```

Note that using a readtable, you can implement languages that go beyond plain S-expressions.

In addition to this wrapper, there is also `#:wrapper2` that has more control over the resulting reader functions. If specified, this wrapper is handed the input port and a (one-argumet) reader function that expects the input port as an argument. This allows this wrapper to hand a different port value to the reader function, for example, it can divert the read to use different file (if given a port that corresponds to a file). Here is the case-insensitive implemented using this option:

```
(module insensitive syntax/module-reader
  scheme/base
  #:wrapper2 (lambda (in r)
               (parameterize ([read-case-sensitive #f])
                 (r in))))
```

In some cases, the reader functions read the whole file, so there is no need to iterate them (e.g., `read-inside` and `read-syntax-inside`). In these cases you can specify `#:whole-body-readers?` as `#t` — the readers are expected to return a list of expressions in this case.

Finally, note that the two wrappers can return a different value than the wrapped function. This introduces two more customization points for the resulting readers:

- The thunk that is passed to a `#:wrapper1` function reads the file contents and returns a list of read expressions (either syntax values or S-expressions). For example, the following reader defines a "language" that ignores the contents of the file, and simply reads files as if they were empty:

```
(module ignored syntax/module-reader
  scheme/base
  #:wrapper1 (lambda (t) (t) '()))
```

  Note that it is still performing the read, otherwise the module loader will complain about extra expressions.

- The reader function that is passed to a `#:wrapper2` function returns the final reault of the reader (a module expression). You can return a different value, for example, making it use a different language module.

In some rare cases, it is more convenient to know whether a reader is invoked for a `read` or for a `read-syntax`. To accommodate these cases, both wrappers can accept an additional argument, and in this case, they will be handed a boolean value that indicates whether the reader is expected to read syntax (`#t`) or not (`#f`). For example, here is a reader that uses the scribble syntax, and the first datum in the file determines the actual language (which means that the library specification is effectively ignored):

```
(module scribbled syntax/module-reader
  -ignored-
  #:wrapper2
  (lambda (in rd stx?)
    (let* ([lang (read in)]
           [mod  (parameterize ([current-readtable (make-at-readtable)])
                   (rd in))]
           [mod  (if stx? mod (datum->syntax #f mod))]
           [r (syntax-case mod ()
                [(module name lang* . body)
                 (with-syntax ([lang (datum->syntax
                                      #'lang* lang #'lang*)])
                   (syntax/loc mod (module name lang . body)))])])
      (if stx? r (syntax->datum r))))
  (require scribble/reader))
```

```
(wrap-read-all mod-path
               in
               read
               mod-path-stx
               src
               line
               col
               pos)          → any/c
  mod-path : module-path?
  in : input-port?
  read : (input-port . -> . any/c)
  mod-path-stx : syntax?
  src : (or/c syntax? #f)
  line : number?
  col : number?
  pos : number?
```

[Note: this function is deprecated; `syntax/module-reader` can be adapted using the various keywords to arbitrary readers, please use it instead.]

Repeatedly calls *read* on *in* until an end of file, collecting the results in order into *lst*, and derives a *name-id* from `(object-name in)`. The last five arguments are used to construct the syntax object for the language position of the module. The result is roughly

```
`(module ,name-id ,mod-path ,@lst)
```

5 Non-Module Compilation And Expansion

```
(require syntax/toplevel)
```

```
(expand-syntax-top-level-with-compile-time-evals stx) → syntax?
  stx : syntax?
```

Expands *stx* as a top-level expression, and evaluates its compile-time portion for the benefit of later expansions.

The expander recognizes top-level `begin` expressions, and interleaves the evaluation and expansion of of the `begin` body, so that compile-time expressions within the `begin` body affect later expansions within the body. (In other words, it ensures that expanding a `begin` is the same as expanding separate top-level expressions.)

The *stx* should have a context already, possibly introduced with `namespace-syntax-introduce`.

```
(expand-top-level-with-compile-time-evals stx) → syntax?
  stx : syntax?
```

Like `expand-syntax-top-level-with-compile-time-evals`, but *stx* is first given context by applying `namespace-syntax-introduce` to it.

```
(expand-syntax-top-level-with-compile-time-evals/flatten stx)
 → (listof syntax?)
  stx : syntax?
```

Like `expand-syntax-top-level-with-compile-time-evals`, except that it returns a list of syntax objects, none of which have a `begin`. These syntax objects are the flattened out contents of any `begin`s in the expansion of *stx*.

```
(eval-compile-time-part-of-top-level stx) → void?
  stx : syntax?
```

Evaluates expansion-time code in the fully expanded top-level expression represented by *stx* (or a part of it, in the case of `begin` expressions). The expansion-time code might affect the compilation of later top-level expressions. For example, if *stx* is a `require` expression, then `namespace-require/expansion-time` is used on each require specification in the form. Normally, this function is used only by `expand-top-level-with-compile-time-evals`.

```
(eval-compile-time-part-of-top-level/compile stx)
 → (listof compiled-expression?)
  stx : syntax?
```

Like `eval-compile-time-part-of-top-level`, but the result is compiled code.

6 Trusting Standard Recertifying Transformers

```
(require syntax/trusted-xforms)
```

The syntax/trusted-xforms library has no exports. It exists only to require other modules that perform syntax transformations, where the other transformations must use syntax-recertify. An application that wishes to provide a less powerful code inspector to a sub-program should generally attach syntax/trusted-xforms to the sub-program's namespace so that things like the class system from scheme/class work properly.

7 Attaching Documentation to Exports

```
(require syntax/docprovide)
```

```
(provide-and-document doc-label-id doc-row ...)

doc-row = (section-string (name type-datum doc-string ...) ...)
        | (all-from prefix-id module-path doc-label-id)
        | (all-from-except prefix-id module-path doc-label-id id ...)

   name = id
        | (local-name-id external-name-id)
```

A form that exports names and records documentation information.

The *doc-label-id* identifier is used as a key for accessing the documentation through `lookup-documentation`. The actual documentation is organized into "rows", each with a section title.

A `row` has one of the following forms:

- `(section-string (name type-datum doc-string ...) ...)`

  Creates a documentation section whose title is *section-string*, and provides/documents each *name*. The *type-datum* is arbitrary, for use by clients that call `lookup-documentation`. The *doc-string*s are also arbitrary documentation information, usually concatenated by clients.

  A *name* is either an identifier or a renaming sequence `(local-name-id extenal-name-id)`.

  Multiple `rows` with the same section name will be merged in the documentation output. The final order of sections matches the order of the first mention of each section.

- `(all-from prefix-id module-path doc-label-id)`
- `(all-from-except prefix-id module-path doc-label-id id ...)`

  Merges documentation and provisions from the specified module into the current one; the *prefix-id* is used to prefix the imports into the current module (so they can be re-exported). If *id*s are provided, the specified *id*s are not re-exported and their documentation is not merged.

```
(lookup-documentation module-path-v
                      label-sym)    → any
  module-path-v : module-path?
  label-sym : symbol?
```

Returns documentation for the specified module and label. The *module-path-v* argument is a quoted module path, like the argument to dynamic-require. The *label-sym* identifies a set of documentation using the symbol as a label identifier in provide-and-document.

Index