

Syntax: Meta-Programming Helpers

Version 4.2.1

July 30, 2009

Contents

1	Syntax Object Helpers	4
1.1	Deconstructing Syntax Objects	4
1.2	Matching Fully-Expanded Expressions	5
1.3	Hashing on <code>bound-identifier=?</code> and <code>free-identifier=?</code>	6
1.4	Identifier dictionaries	9
1.4.1	Dictionaries for <code>bound-identifier=?</code>	9
1.4.2	Dictionaries for <code>free-identifier=?</code>	12
1.5	Rendering Syntax Objects with Formatting	13
1.6	Computing the Free Variables of an Expression	13
1.7	Replacing Lexical Context	13
1.8	Legacy Zodiac Interface	14
2	Module-Processing Helpers	15
2.1	Reading Module Source Code	15
2.2	Getting Module Compiled Code	15
2.3	Resolving Module Paths to File Paths	17
2.4	Simplifying Module Paths	18
2.5	Inspecting Modules and Module Dependencies	18
3	Macro Transformer Helpers	20
3.1	Extracting Inferred Names	20
3.2	Support for <code>local-expand</code>	20
3.3	Parsing <code>define</code> -like Forms	20
3.4	Flattening <code>begin</code> Forms	21
3.5	Expanding <code>define-struct</code> -like Forms	21

3.6	Resolving <code>include</code> -like Paths	25
3.7	Controlling Syntax Templates	26
4	Reader Helpers	28
4.1	Raising <code>exn:fail:read</code>	28
4.2	Module Reader	29
5	Non-Module Compilation And Expansion	34
6	Trusting Standard Recertifying Transformers	36
7	Attaching Documentation to Exports	37
8	Parsing and classifying syntax	39
8.1	Parsing syntax	39
8.1.1	S-pattern variants	41
8.1.2	H-pattern variants	47
8.1.3	EH-pattern forms	48
8.1.4	Pattern directives	49
8.2	Syntax Classes	51
8.2.1	Attributes	53
8.2.2	Inspection tools	53
8.3	Literal sets and Conventions	54
8.4	Library syntax classes and literal sets	55
8.4.1	Syntax classes	55
8.4.2	Literal sets	56
	Index	57

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 flattening 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 `clauses` 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 (extra-id ...) 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-syntax-case/phase stx-expr phase-expr clause ...)
```

Generalizes `kernel-syntax-case` to work at an arbitrary phase level, as indicated by `phase-expr`.

```
(kernel-syntax-case*/phase stx-expr phase-expr (extra-id ..)
  clause ...)
```

Generalizes `kernel-syntax-case*` to work at an arbitrary phase level, as indicated by `phase-expr`.

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

Returns a list of identifiers that are bound normally, `for-syntax`, and `for-template` to the primitive PLT Scheme forms for expressions and internal-definition positions (so the list does not include `#:require` or `#:provide`). 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=?`

See also `syntax/id-table` for an implementation of identifier mappings using the `scheme/dict` dictionary interface.

```
(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 : bound-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-free-identifier-mapping`, etc.

1.4 Identifier dictionaries

```
(require syntax/id-table)
```

This module provides functionality like that of `syntax/boundmap` but with more operations, standard names, implementation of the `scheme/dict` interface, and immutable (functionally-updating) variants.

1.4.1 Dictionaries for `bound-identifier=?`

Bound-identifier tables implement the dictionary interface of `scheme/dict`. Consequently, all of the appropriate generic functions (`dict-ref`, `dict-map`, etc) can be used on free-identifier tables.

```
(make-bound-id-table [init-dict]) → mutable-bound-id-table?
  init-dict : dict? = null
(make-immutable-bound-id-table [init-dict])
→ immutable-bound-id-table?
  init-dict : dict? = null
```

Produces a dictionary mapping syntax identifiers to arbitrary values. The mapping uses `bound-identifier=?` to compare keys, but also uses a hash table based on symbol equality to make the mapping efficient in the common case. The two procedures produce mutable and immutable dictionaries, respectively.

The optional `init-dict` argument provides the initial mappings. It must be a dictionary, and its keys must all be identifiers. If the `init-dict` dictionary has multiple distinct entries whose keys are `bound-identifier=?`, only one of the entries appears in the new id-table, and it is not specified which entry is picked.

```
(bound-id-table? v) → boolean?
  v : any/c
```

Returns `#t` if `v` was produced by `make-bound-id-table` or `make-immutable-bound-id-table`, `#f` otherwise.

```
(mutable-bound-id-table? v) → boolean?
  v : any/c
(immutable-bound-id-table? v) → boolean?
  v : any/c
```

Predicate for the mutable and immutable variants of bound-identifier tables, respectively.

```
(bound-id-table-ref table id [failure]) → any
  table : bound-id-table?
  id : identifier?
  failure : any/c = (lambda () (raise (make-exn:fail ....)))
```

Like `hash-ref` for bound identifier tables. In particular, if `id` is not found, the `failure` argument is applied if it is a procedure, or simply returned otherwise.

```
(bound-id-table-set! table id v) → void?
  table : mutable-bound-id-table?
  id : identifier?
  v : any/c
```

Like `hash-set!` for mutable bound-identifier tables.

```
(bound-id-table-set table id v) → immutable-bound-id-table?  
  table : immutable-bound-id-table?  
  id : identifier?  
  v : any/c
```

Like `hash-set` for immutable bound-identifier tables.

```
(bound-id-table-remove! table id) → void?  
  table : mutable-bound-id-table?  
  id : identifier?
```

Like `hash-remove!` for mutable bound-identifier tables.

```
(bound-id-table-remove table id v) → immutable-bound-id-table?  
  table : immutable-bound-id-table?  
  id : identifier?  
  v : any/c
```

Like `hash-remove` for immutable bound-identifier tables.

```
(bound-id-table-map table proc) → list?  
  table : bound-id-table?  
  proc : (-> identifier? any/c any)
```

Like `hash-map` for bound-identifier tables.

```
(bound-id-table-for-each table proc) → void?  
  table : bound-id-table?  
  proc : (-> identifier? any/c any)
```

Like `hash-for-each` for bound-identifier tables.

```
(bound-id-table-count table) → exact-nonnegative-integer?  
  table : bound-id-table?
```

Like `hash-count` for bound-identifier tables.

1.4.2 Dictionaries for `free-identifier=?`

Free-identifier tables implement the dictionary interface of `scheme/dict`. Consequently, all of the appropriate generic functions (`dict-ref`, `dict-map`, etc) can be used on free-identifier tables.

```
(make-free-id-table [init-dict]) → mutable-free-id-table?
  init-dict : dict? = null
(make-immutable-free-id-table [init-dict])
→ immutable-free-id-table?
  init-dict : dict? = null
(free-id-table? v) → boolean?
  v : any/c
(mutable-free-id-table? v) → boolean?
  v : any/c
(immutable-free-id-table? v) → boolean?
  v : any/c
(free-id-table-ref table id [failure]) → any
  table : free-id-table?
  id : identifier?
  failure : any/c = (lambda () (raise (make-exn:fail ....)))
(free-id-table-set! table id v) → void?
  table : mutable-free-id-table?
  id : identifier?
  v : any/c
(free-id-table-set table id v) → immutable-free-id-table?
  table : immutable-free-id-table?
  id : identifier?
  v : any/c
(free-id-table-remove! table id) → void?
  table : mutable-free-id-table?
  id : identifier?
(free-id-table-remove table id v) → immutable-free-id-table?
  table : immutable-free-id-table?
  id : identifier?
  v : any/c
(free-id-table-map table proc) → list?
  table : free-id-table?
  proc : (-> identifier? any/c any)
(free-id-table-for-each table proc) → void?
  table : free-id-table?
  proc : (-> identifier? any/c any)
(free-id-table-count table) → exact-nonnegative-integer?
  table : free-id-table?
```

Like the procedures for bound-identifier tables ([make-bound-id-table](#), [bound-id-table-ref](#), etc), but for free-identifier tables, which use [free-identifier=?](#) to compare keys.

1.5 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*.

1.6 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 (see §1.2.3.1 “Fully Expanded Programs” and [expand](#)).

1.7 Replacing Lexical Context

```
(require syntax/strip-context)
```

```
(strip-context stx) → syntax?  
stx : syntax?
```

Removes all lexical context from *stx*, preserving source-location information and properties.

```
(replace-context ctx-stx stx) → syntax?  
ctx-stx : (or/c syntax? #f)  
stx : syntax?
```

Uses the lexical context of `ctx-stx` to replace the lexical context of all parts of `stx`, preserving source-location information and properties of `stx`.

1.8 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 preferred 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 Flattening `begin` Forms

```
(require syntax/flatten-begin)
```

```
(flatten-begin stx) → (listof syntax?)
stx : syntax?
```

Extracts the sub-expressions from a `begin`-like form, reporting an error if `stx` does not have the right shape (i.e., a syntax list). The resulting syntax objects have annotations transferred from `stx` using `syntax-track-origin`.

3.5 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.

3.6 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.

3.7 Controlling Syntax Templates

```
(require syntax/template)
```

```
(transform-template template-stx
  #:save save-proc
  #:restore-stx restore-proc-stx
  [#:leaf-save leaf-save-proc
   #:leaf-restore-stx leaf-restore-proc-stx
   #:leaf-datum-stx leaf-datum-proc-stx
   #:pvar-save pvar-save-proc
   #:pvar-restore-stx pvar-restore-stx
   #:cons-stx cons-proc-stx
   #:ellipses-end-stx ellipses-end-stx
   #:constant-as-leaf? constant-as-leaf?])
→ syntax?
template-stx : syntax?
save-proc : (syntax? . -> . any/c)
restore-proc-stx : syntax?
leaf-save-proc : (syntax? . -> . any/c) = save-proc
leaf-restore-proc-stx : syntax? = #'(lambda (data stx) stx)
leaf-datum-proc-stx : syntax? = #'(lambda (v) v)
pvar-save-proc : (identifier? . -> . any/c) = (lambda (x) #f)
pvar-restore-stx : syntax? = #'(lambda (d stx) stx)
cons-proc-stx : syntax? = cons
ellipses-end-stx : syntax? = #'values
constant-as-leaf? : boolean? = #f
```

Produces an representation of an expression similar to #'(syntax #, *template-stx*), but functions like *save-proc* can collect information that might otherwise be lost by syntax (such as properties when the syntax object is marshaled within bytecode), and run-time functions like the one specified by *restore-proc-stx* can use the saved information or otherwise process the syntax object that is generated by the template.

The *save-proc* is applied to each syntax object in the representation of the original template (i.e., in *template-stx*). If *constant-as-leaf?* is #t, then *save-proc* is applied only to syntax objects that contain at least one pattern variable in a sub-form. The result of *save-proc* is provided back as the first argument to *restore-proc-stx*, which indicates a function with a contract (-> any/c syntax any/c any/c); the second argument to *restore-proc-stx* is the syntax object that syntax generates, and the last argument is a datum that have been processed recursively (by functions such as *restore-proc-stx*) and that normally would be converted back to a syntax object using the second argument's context, source, and properties. Note that *save-proc* works at expansion time (with respect to the template form), while *restore-proc-stx* indicates a function that is called at run time (for the template form), and the data that flows from *save-proc* to *restore-proc-*

`stx` crosses phases via `quote`.

The `leaf-save-proc` and `leaf-restore-proc-stx` procedures are analogous to `save-proc` and `restore-proc-stx`, but they are applied to leaves, so there is no third argument for recursively processed sub-forms. The function indicated by `leaf-restore-proc-stx` should have the contract `(-> any/c syntax? any/c)`.

The `leaf-datum-proc-stx` procedure is applied to leaves that are not syntax objects, which can happen because pairs and the empty list are not always individually wrapped as syntax objects. The function should have the contract `(-> any/c any/c)`. When `constant-as-leaf?` is `#f`, the only possible argument to the procedure is `null`.

The `pvar-save` and `pvar-restore-stx` procedures are analogous to `save-proc` and `restore-proc-stx`, but they are applied to pattern variables. The `pvar-restore-stx` procedure should have the contract `(-> any/c syntax? any/c)`, where the second argument corresponds to the substitution of the pattern variable.

The `cons-proc-stx` procedure is used to build intermediate pairs, including pairs passed to `restore-proc-stx` and pairs that do not correspond to syntax objects.

The `ellipses-end-stx` procedure is an extra filter on the syntax object that follows a sequence of `...` ellipses in the template. The procedure should have the contract `(-> any/c any/c)`.

The following example illustrates a use of `transform-template` to implement a `syntax/shape` form that preserves the `'paren-shape` property from the original template, even if the template code is marshaled within bytecode.

```
(define-for-syntax (get-shape-prop stx)
  (syntax-property stx 'paren-shape))

(define (add-shape-prop v stx datum)
  (syntax-property (datum->syntax stx datum stx stx)
                   'paren-shape
                   v))

(define-syntax (syntax/shape stx)
  (syntax-case stx ()
    [(_ tmpl)
     (transform-template #'tmpl
                         #:save get-shape-prop
                         #:restore-stx #'add-shape-prop)]))
```

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

reader-option = #:language lang-expr
                | #: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 reader 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 reader syntax/module-reader
  scheme/base
  #:wrapper1 (lambda (t)
    (parameterize ([read-case-sensitive #f])
      (t))))
```

Note that using a readable, you can implement languages that are extensions of 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-argument) 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 reader 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., Scribble’s `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.

In addition, 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 result 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 reader 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))

```

This ability to change the language position in the resulting module expression can be useful in cases such as the above, where the base language module is chosen based on the input. To make this more convenient, you can omit the *module-path* and instead specify it via a `#:language` expression. This expression can evaluate to a datum which is used as a language, or it can evaluate to a thunk. In the latter case, the thunk will be invoked to return such a datum before reading the module body begins, in a dynamic extent where `current-input-port` is the source input. Using this, the last example above can be written more concisely:

```

(module reader syntax/module-reader
  #:language read
  #:wrapper2 (lambda (in rd stx?)
               (parameterize ([current-readtable (make-at-readtable)])
                 (rd in)))
  (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 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 `begins` 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-strings` are also arbitrary documentation information, usually concatenated by clients.
A `name` is either an identifier or a renaming sequence (`local-name-id external-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 `ids` are provided, the specified `ids` 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`.

8 Parsing and classifying syntax

The `syntax/parse` library provides a framework for describing and parsing syntax. Using `syntax/parse`, macro writers can define new syntactic categories, specify their legal syntax, and use them to write clear, concise, and robust macros. The library also provides a pattern-matching form, `syntax-parse`, which offers many improvements over `syntax-case`.

```
(require syntax/parse)
```

8.1 Parsing syntax

This section describes the `syntax-parse` pattern matching form, syntax patterns, and attributes.

```
(syntax-parse stx-expr parse-option ... clause ...+)

parse-option = #:literals (literal ...)
               | #:literal-sets (literal-set ...)
               | #:conventions (convention-id ...)

               literal = literal-id
                       | (pattern-id literal-id)

               literal-set = literal-set-id
                             | [literal-set-id #:at context-id]

               clause = (syntax-pattern pattern-directive ... expr)
```

Evaluates `stx-expr`, which should produce a syntax object, and matches it against the `clauses` in order. If some clause's pattern matches, its attributes are bound to the corresponding subterms of the syntax object and that clause's side conditions and `expr` is evaluated. The result is the result of `expr`.

If the syntax object fails to match any of the patterns (or all matches fail the corresponding clauses' side conditions), a syntax error is raised.

The `#:literals` option specifies identifiers that should match as literals, rather than simply being pattern variables. A literal in the literals list has two components: the identifier used within the pattern to signify the positions to be matched (`pattern-id`), and the identifier expected to occur in those positions (`literal-id`). If the single-identifier form is used, the same identifier is used for both purposes.

Note: Unlike `syntax-case`, `syntax-parse` requires all literals to have a binding. To match

identifiers by their symbolic names, consider using the `atom-in-list` syntax class instead.

Many literals can be declared at once via one or more literal sets, imported with the `#:literal-sets` option. The literal-set definition determines the literal identifiers to recognize and the names used in the patterns to recognize those literals.

The `#:conventions` option imports conventions that give default syntax classes to pattern variables that do not explicitly specify a syntax class.

```
(syntax-parser maybe-literals clause ...)
```

Like `syntax-parse`, but produces a matching procedure. The procedure accepts a single argument, which should be a syntax object.

The grammar of *syntax patterns* accepted by `syntax-parse` and `syntax-parser` is given in the following table:

```
S-pattern = pvar-id
            | pvar-id:syntax-class-id
            | literal-id
            | atomic-datum
            | (H-pattern . S-pattern)
            | ((~or EH-pattern ...+) ... . S-pattern)
            | (EH-pattern ... . S-pattern)
            | (~and S-pattern ...+)
            | (~or S-pattern ...+)
            | #(pattern-part ...)
            | #s(prefab-struct-key pattern-part ...)
            | (~rest S-pattern)
            | (~describe expr S-pattern)
            | (~! . S-pattern)
            | (~bind [attr-id expr] ...)
            | (~fail maybe-fail-condition message-expr)

L-pattern = ()
            | (H-pattern . L-pattern)
            | ((~or EH-pattern ...+) ... . L-pattern)
            | (EH-pattern ... . L-pattern)
            | (~rest L-pattern)
            | (~! . L-pattern)

H-pattern = (~or H-pattern ...+)
            | (~seq . L-pattern)
            | (~describe expr H-pattern)
            | S-pattern
```

```

EH-pattern = (~once H-pattern once-option ...)
              | (~optional H-pattern optional-option ...)
              | H-pattern

```

There are three main kinds of syntax pattern: S-patterns (for “single patterns”), H-patterns (for “head patterns”), and EH-patterns (for “ellipsis head patterns”). A fourth kind, L-patterns (for “list patterns”), is a restricted subset of S-patterns. When a special form in this manual refers to *syntax-pattern* (eg, the description of the `syntax-parse` special form), it means specifically S-pattern.

8.1.1 S-pattern variants

An *S-pattern* (for “single pattern”) is a pattern that describes a single term. The pattern may, of course, consist of other parts. For example, `(17 ...)` is an S-pattern that matches any term that is a proper list of repeated 17 numerals. The *L-patterns* (for “list pattern”) are S-pattern having a restricted structure that constrains it to match only terms that are proper lists.

Here are the variants of S-pattern:

pvar-id

If *pvar-id* has no syntax class (by `#:declare` or `#:convention`), the pattern matches anything. The pattern variable is bound to the matched subterm, unless the pattern variable is the wildcard (`_`), in which case no binding occurs.

If *pvar-id* does have an associated syntax class, it behaves like the following form.

pvar-id:syntax-class-id

Matches only subterms specified by the *syntax-class-id*. The syntax class’s attributes are computed for the subterm and bound to the pattern variables formed by prefixing *pvar-id*. to the name of the attribute. *pvar-id* is bound to the matched subterm.

If *pvar-id* is `_`, no attributes are bound.

If *pvar-id* is empty (that is, if the pattern is of the form `:syntax-class-id`), then the syntax class’s attributes are bound, but their names are not prefixed first.

Examples:

```
> (syntax-parse #'x
  [var:id (syntax-e #'var)])
x
> (syntax-parse #'12
  [var:id (syntax-e #'var)])
eval:2:0: ?: expected identifier at: 12 in: 12
> (syntax-parse #'(x y z)
  [var:id (syntax-e #'var)])
eval:3:0: x: expected identifier at: (x y z) in: (x y z)
```

literal-id

An identifier that appears in the literals list is not a pattern variable; instead, it is a literal that matches any identifier `free-identifier=?` to it.

Specifically, if *literal-id* is the “pattern” name of an entry in the literals list, then it represents a pattern that matches only identifiers `free-identifier=?` to the “literal” name. These identifiers are often the same.

Examples:

```
> (syntax-parse #'(define x 12)
  #:literals (define)
  [(define var:id body:expr) 'ok])
ok
> (syntax-parse #'(lambda x 12)
  #:literals (define)
  [(define var:id body:expr) 'ok])
eval:5:0: lambda: expected the literal identifier define
at: lambda in: (lambda x 12)
> (syntax-parse #'(define x 12)
  #:literals ([def define])
  [(def var:id body:expr) 'ok])
ok
> (syntax-parse #'(lambda x 12)
  #:literals ([def define])
  [(def var:id body:expr) 'ok])
eval:7:0: lambda: expected the literal identifier define
at: lambda in: (lambda x 12)
```

atomic-datum

Numbers, strings, booleans, keywords, and the empty list match as literals.

Examples:

```
> (syntax-parse #'(a #:foo bar)
  [(x #:foo y) (syntax->datum #'y)])
bar
> (syntax-parse #'(a foo bar)
  [(x #:foo y) (syntax->datum #'y)])
eval:9:0: a: expected the literal #:foo at: foo in: (a foo
bar)
```

(H-pattern . S-pattern)

Matches any term that can be decomposed into a list prefix matching the H-pattern and a suffix matching the S-pattern.

Note that the pattern may match terms that are not even improper lists; if the head pattern can match a zero-length head, then the whole pattern matches whatever the tail pattern accepts.

The first pattern can be an S-pattern, in which case the whole pattern matches any pair whose first element matches the first pattern and whose rest matches the second.

See H-patterns for more information.

((~or EH-pattern ...+) S-pattern)

(EH-pattern S-pattern)

Matches any term that can be decomposed into a list head matching some number of repetitions of the EH-pattern alternatives (subject to its repetition constraints) followed by a list tail matching the S-pattern.

In other words, the whole pattern matches either the second pattern (which need not be a list) or a term whose head matches one of the alternatives of the first pattern and whose tail recursively matches the whole sequence pattern.

The ~or-free variant is shorthand for the ~or variant with just one alternative.

See EH-patterns for more information.

(~and S-pattern ...)

Matches any syntax that matches all of the included patterns.

Attributes bound in subpatterns are available to subsequent subpatterns. The whole pattern binds all of the subpatterns' attributes.

One use for `~and`-patterns is preserving a whole term (including its lexical context, source location, etc) while also examining its structure. Syntax classes are useful for the same purpose, but `~and` can be lighter weight.

Example:

```
> (syntax-parse #'(m (import one two))
  #:literals (import)
  [(~ ( ~and import-clause (import i ...)))
    (let ([bad (check-imports
                 (syntax->list #'(i ...)))]
          (when bad
            (raise-syntax-error
             #f "bad import" #'import-clause bad)
            'ok))])
  ])
```

?: unbound literal not allowed

`(~or S-pattern ...)`

Matches any term that matches one of the included patterns.

The whole pattern binds *all* of the subpatterns' attributes. An attribute that is not bound by the "chosen" subpattern has a value of `#f`. The same attribute may be bound by multiple subpatterns, and if it is bound by all of the subpatterns, it is sure to have a value if the whole pattern matches.

Examples:

```
> (syntax-parse #'a
  [(~or x:id (~and x #f)) (syntax->datum #'x)])
a
> (syntax-parse #'#f
  [(~or x:id (~and x #f)) (syntax->datum #'x)])
#f
```

`#(pattern-part ...)`

Matches a term that is a vector whose elements, when considered as a list, match the S-pattern corresponding to `(pattern-part ...)`.

Examples:

```
> (syntax-parse #'#(1 2 3)
  [#(x y z) (syntax->datum #'z)])
3
> (syntax-parse #'#(1 2 3)
  [#(x y ...) (syntax->datum #'(y ...))])
(2 3)
> (syntax-parse #'#(1 2 3)
  [#(x ~rest y) (syntax->datum #'y)])
(2 3)
```

`#s(prefab-struct-key pattern-part ...)`

Matches a term that is a prefab struct whose key is exactly the given key and whose sequence of fields, when considered as a list, match the S-pattern corresponding to `(pattern-part ...)`.

Examples:

```
> (syntax-parse #'#s(point 1 2 3)
    [#s(point x y z) 'ok])
ok
> (syntax-parse #'#s(point 1 2 3)
    [#s(point x y ...) (syntax->datum #'(y ...))])
(2 3)
> (syntax-parse #'#s(point 1 2 3)
    [#s(point x ~rest y) (syntax->datum #'y)])
(2 3)
```

`(~rest S-pattern)`

Matches just like the inner *S-pattern*. The `~rest` pattern form is useful in positions where improper lists (“dots”) are not allowed by the reader, such as vector and structure patterns (see above).

Examples:

```
> (syntax-parse #'(1 2 3)
    [(x ~rest y) (syntax->datum #'y)])
(2 3)
> (syntax-parse #'#(1 2 3)
    [#(x ~rest y) (syntax->datum #'y)])
(2 3)
```

`(~describe expr S-pattern)`

The `~describe` pattern form annotates a pattern with a description, a string expression that is evaluated in the scope of all prior attribute bindings. If parsing the inner pattern fails, then the description is used to synthesize the error message.

A describe-pattern also affects backtracking in two ways:

- A cut-pattern (`~!`) within a describe-pattern only eliminates choice-points created within the describe-pattern.
- If a describe-pattern succeeds, then all choice points created within the describe-pattern are discarded, and a failure *after* the describe-pattern backtracks to a choice point *before* the describe-pattern, never one *within* it.

`(~! . S-pattern)`

The `~!` operator, pronounced “cut”, eliminates backtracking choice points and commits parsing to the current branch of the pattern it is exploring.

Common opportunities for cut-patterns come from recognizing special forms based on keywords. Consider the following expression:

```
> (syntax-parse #'(define-values a 123)
    #:literals (define-values define-syntaxes)
    [(define-values (x:id ...) e) 'define-values]
    [(define-syntaxes (x:id ...) e) 'define-syntaxes]
    [e 'expression])
expression
```

Given the ill-formed term `(define-values a 123)`, the expression tries the first clause, fails to match `a` against the pattern `(x:id ...)`, and then backtracks to the second clause and ultimately the third clause, producing the value `'expression`. But the term is not an expression; it is an ill-formed use of `define-values`! The proper way to write the `syntax-parse` expression follows:

```
> (syntax-parse #'(define-values a 123)
    #:literals (define-values define-syntaxes)
    [(define-values ~! (x:id ...) e) 'define-values]
    [(define-syntaxes ~! (x:id ...) e) 'define-syntaxes]
    [e 'expression])
eval:23:0: define-values: expected sequence of terms or
expected the literal () at: a in: (define-values a 123)
```

Now, given the same term, `syntax-parse` tries the first clause, and since the keyword `define-values` matches, the cut-pattern commits to the current pattern, eliminating the choice points for the second and third clauses. So when the clause fails to match, the `syntax-parse` expression raises an error.

The effect of a `~!` pattern is delimited by the nearest enclosing `~describe` pattern. If there is no enclosing `~describe` pattern but the cut occurs within a syntax class definition, then only choice points within the syntax class definition are discarded.

`(~bind [attr-id expr] ...)`

This pattern matches any term. Its effect is to evaluate the `exprs` and bind them to the given `attr-ids` as attributes.

```
(~fail maybe-fail-condition message-expr)
```

```
maybe-fail-condition =  
    | #:when condition-expr  
    | #:unless condition-expr
```

This pattern succeeds or fails independent of the term being matched against. If the condition is absent, or if the `#:when` condition evaluates to a true value, or if the `#:unless` condition evaluates to `#f`, then the pattern fails with the given message. Otherwise the pattern succeeds.

Fail patterns can be used together with cut patterns to recognize specific ill-formed terms and address them with specially-created failure messages.

8.1.2 H-pattern variants

An *H-pattern* (for “head pattern”) is a pattern that describes some number of terms that occur at the head of some list (possibly an improper list). An H-pattern’s usefulness comes from being able to match heads of different lengths. H-patterns are useful for specifying optional forms such as keyword arguments.

Here are the variants of H-pattern:

```
(~seq . L-pattern)
```

Matches a head whose elements, if put in a list, would match the given L-pattern.

Example:

```
> (syntax-parse #'(1 2 3 4)  
   [((~seq 1 2 3) 4) 'ok])  
ok
```

```
(~or H-pattern ...)
```

Like the S-pattern version of `~or`, but matches a term head instead.

Example:

```
> (syntax-parse #'(:foo 2 a b c)  
   [((~or (~seq #:foo x) (~seq)) y:id ...)  
    (attribute x)])  
#<syntax:25:0>
```

```
(~describe expr H-pattern)
```

Like the S-pattern version of `~describe`, but matches a head pattern instead.

S-pattern

Matches a head of one element, which must be a term matching the given S-pattern.

8.1.3 EH-pattern forms

An *EH-pattern* (for “ellipsis-head pattern”) is pattern that describes some number of terms, like an H-pattern, but may also place constraints on the number of times it occurs in a repetition. EH-patterns (and ellipses) are useful for matching keyword arguments where the keywords may come in any order.

Examples:

```
> (define parser1
  (syntax-parser
    [((~or (~once (~seq #:a x) #:name "#:a keyword")
             (~optional (~seq #:b y) #:name "#:b keyword")
             (~seq #:c z)) ...)
      'ok]))
> (parser1 #'( #:a 1))
ok
> (parser1 #'( #:b 2 #:c 3 #:c 25 #:a 'hi))
ok
> (parser1 #'( #:a 1 #:a 2))
eval:29:0: ?: too many occurrences of #:a keyword after 4
terms at: ( #:a 1 #:a 2) in: ( #:a 1 #:a 2)
```

The pattern requires exactly one occurrence of the `#:a` keyword and argument, at most one occurrence of the `#:b` keyword and argument, and any number of `#:c` keywords and arguments. The “pieces” can occur in any order.

Here are the variants of EH-pattern:

```
(~once H-pattern once-option ...)
```

```
once-option = #:name name-expr
              | #:too-few too-few-message-expr
              | #:too-many too-many-message-expr
```

Matches if the inner H-pattern matches. This pattern must be selected exactly once in the match of the entire repetition sequence.

If the pattern is not chosen in the repetition sequence, then an error is raised with a message, either *too-few-message-expr* or "missing required occurrence of *name-expr*".

If the pattern is chosen more than once in the repetition sequence, then an error is raised with a message, either *too-many-message-expr* or "too many occurrences of *name-expr*".

```
(~optional H-pattern optional-option ...)
```

```
optional-option = #:name name-expr  
                | #:too-many too-many-message-expr
```

Matches if the inner H-pattern matches. This pattern may be used at most once in the match of the entire repetition.

If the pattern is chosen more than once in the repetition sequence, then an error is raised with a message, either *too-many-message-expr* or "too many occurrences of *name-expr*".

8.1.4 Pattern directives

Both `syntax-parse` and `syntax-parser` support directives for annotating the pattern and specifying side conditions. The grammar for pattern directives follows:

```
pattern-directive = #:declare pattern-id syntax-class-id  
                  | #:declare pattern-id (syntax-class-id expr ...)  
                  | #:with syntax-pattern expr  
                  | #:fail-when condition-expr message-expr  
                  | #:fail-unless condition-expr message-expr
```

```
#:declare pvar-id syntax-class-id
```

```
#:declare pvar-id (syntax-class-id expr ...)
```

The first form is equivalent to using the *pvar-id:syntax-class-id* form in the pattern (but it is illegal to use both for a single pattern variable). The `#:declare` form may be preferred when writing macro-defining macros or to avoid dealing with structured identifiers.

The second form allows the use of parameterized syntax classes, which cannot be expressed using the “colon” notation. The *expr*s are evaluated outside the scope of any of the attribute bindings from pattern that the `#:declare` directive applies to.

```
#:with syntax-pattern expr
```

Evaluates the *expr* in the context of all previous attribute bindings and matches it against the pattern. If the match succeeds, the pattern's attributes are added to environment for the evaluation of subsequent side conditions. If the `#:with` match fails, the matching process backtracks. Since a syntax object may match a pattern in several ways, backtracking may cause the same clause to be tried multiple times before the next clause is reached.

```
#:fail-when condition-expr message-expr
```

```
#:fail-unless condition-expr message-expr
```

Evaluates the *condition-expr* in the context of all previous attribute bindings. If the value is any non-false value for `#:fail-when` or if the value is `#f` for `#:fail-unless`, the matching process backtracks (with the given message); otherwise, it continues.

```
~or  
~and  
~seq  
~once  
~optional  
~rest  
~describe  
~!  
~bind  
~fail
```

Syntax pattern keywords, recognized by `syntax-parse`.

```
(attribute attr-id)
```

Returns the value associated with the attribute named *attr-id*. If *attr-id* is not bound as an attribute, an error is raised. If *attr-id* is an attribute with a nonzero ellipsis depth, then the result has the corresponding level of list nesting.

The values returned by `attribute` never undergo additional wrapping as syntax objects, unlike values produced by some uses of `syntax`, `quasisyntax`, etc. Consequently, the `attribute` form is preferred when the attribute value is used as data, not placed in a syntax object.

8.2 Syntax Classes

Syntax classes provide an abstraction mechanism for the specification of syntax. Built-in syntax classes are supplied that recognize basic classes such as `identifiers` and `keywords`. Programmers can compose basic syntax classes to build specifications of more complex syntax, such as lists of distinct identifiers and formal arguments with keywords. Macros that manipulate the same syntactic structures can share syntax class definitions. The structure of syntax classes and patterns also allows `syntax-parse` to automatically generate error messages for syntax errors.

When a syntax class accepts (matches) a syntax object, it computes and provides attributes based on the contents of the matched syntax. While the values of the attributes depend on the matched syntax, the set of attributes and each attribute's ellipsis nesting depth is fixed for each syntax class.

```
(define-syntax-class name-id stxclass-option ...
  stxclass-variant ...+)
(define-syntax-class (name-id arg-id ...) stxclass-option ...
  stxclass-variant ...+)

stxclass-option = #:attributes (attr-arity-decl ...)
                  | #:description description
                  | #:transparent
                  | #:literals (literal-entry ...)
                  | #:literal-sets (literal-set ...)
                  | #:conventions (convention-id ...)

attr-arity-decl = attr-name-id
                  | (attr-name-id depth)

stxclass-variant = (pattern syntax-pattern stxclass-pattern-directive ...)
```

Defines *name-id* as a syntax class. When the *arg-ids* are present, they are bound as variables (not pattern variables) in the body. The body of the syntax-class definition contains a non-empty sequence of pattern variants.

```
#:attributes (attr-arity-decl ...)
```

Declares the attributes of the syntax class. An attribute arity declaration consists of the attribute name and optionally its ellipsis depth (zero if not explicitly specified).

If the attributes are not explicitly listed, they are inferred as the set of all pattern variables occurring in every variant of the syntax class. Pattern variables that occur at different ellipsis depths are not included, nor are nested attributes.

`#:description` *description*

The *description* argument is an expression (evaluated in a scope containing the syntax class's parameters) that should evaluate to a string. It is used in error messages involving the syntax class. For example, if a term is rejected by the syntax class, an error of the form "expected *description*" may be synthesized.

If absent, the name of the syntax class is used instead.

`#:transparent`

Indicates that errors may be reported with respect to the internal structure of the syntax class.

`#:literals` (*literal-entry*)

`#:literal-sets` (*literal-set ...*)

`#:conventions` (*convention-id ...*)

Declares the literals and conventions that apply to the syntax class's variant patterns and their immediate `#:with` clauses. Patterns occurring within subexpressions of the syntax class (for example, on the right-hand side of a `#:fail-when` clause) are not affected.

These options have the same meaning as under `syntax-parse`.

(`pattern` *syntax-pattern stxclass-pattern-directive ...*)

stxclass-pattern-directive = *pattern-directive*
| `#:rename` *internal-id external-id*

Accepts syntax matching the given syntax pattern with the accompanying pattern directives as in `syntax-parse`.

The attributes of the variant are the attributes of the pattern together with all attributes bound by `#:with` clauses, including nested attributes produced by syntax classes associated with the pattern variables.

```
(define-splicing-syntax-class name-id stxclass-option ...
  stxclass-variant ...+)
(define-splicing-syntax-class (name-id arg-id ...) stxclass-option ...
  stxclass-variant ...+)
```

Defines *name-id* as a splicing syntax class. A splicing syntax class encapsulates H-patterns as an ordinary syntax class encapsulates S-patterns.

pattern

Keyword recognized by `define-syntax-class`. It may not be used as an expression.

8.2.1 Attributes

A syntax class has a set of *attributes*. Each attribute has a name, an ellipsis depth, and a set of nested attributes. When an instance of the syntax class is parsed and bound to a pattern variable, additional pattern variables are bound for each of the syntax class's attributes. The name of these additional pattern variables is the dotted concatenation of the primary pattern variable with the name of the attribute.

For example, if pattern variable `p` is bound to an instance of a syntax class with attribute `a`, then the pattern variable `p.a` is bound to the value of that attribute. The ellipsis depth of `p.a` is the sum of the depths of `p` and attribute `a`.

The attributes of a syntax class are either given explicitly with an `#:attributes` option or inferred from the pattern variables of the syntax class's variants.

8.2.2 Inspection tools

The following special forms are for debugging syntax classes.

```
(syntax-class-attributes syntax-class-id)
```

Returns a list of the syntax class's attributes in flattened form. Each attribute is listed by its name and ellipsis depth.

```
(syntax-class-parse syntax-class-id stx-expr arg-expr ...)
```

Runs the parser for the syntax class (parameterized by the *arg-exprs*) on the syntax object produced by *stx-expr*. On success, the result is a list of vectors representing the attribute

bindings of the syntax class. Each vector contains the attribute name, depth, and associated value. On failure, the result is some internal representation of the failure.

8.3 Literal sets and Conventions

Sometimes the same literals are recognized in a number of different places. The most common example is the literals for fully expanded programs, which are used in many analysis and transformation tools. Specifying literals individually is burdensome and error-prone. As a remedy, `syntax/parse` offers *literal sets*. A literal set is defined via `define-literal-set` and used via the `#:literal-set` option of `syntax-parse`.

```
(define-literal-set name-id (literal ...))
```

```
literal = literal-id  
        | (pattern-id literal-id)
```

Defines *name* as a literal set. Each *literal* can have a separate *pattern-id* and *literal-id*. The *pattern-id* determines what identifiers in the pattern are treated as literals. The *literal-id* determines what identifiers the literal matches.

Examples:

```
> (define-literal-set def-litset  
   (define-values define-syntaxes))  
> (syntax-parse #'(define-syntaxes (x) 12)  
   #:literal-sets (def-litset)  
   [(define-values (x:id ...) e:expr) 'v]  
   [(define-syntaxes (x:id ...) e:expr) 's])  
s
```

```
(define-conventions name-id (id-pattern syntax-class) ...)
```

```
name-pattern = exact-id  
              | name-rx  
  
syntax-class = syntax-class-id  
              | (syntax-class-id expr ...)
```

Defines *conventions* that supply default syntax classes for pattern variables. A pattern variable that has no explicit syntax class is checked against each *id-pattern*, and the first one that matches determines the syntax class for the pattern. If no *id-pattern* matches, then the pattern variable has no syntax class.

Examples:

```

> (define-conventions xyz-as-ids
  [x id] [y id] [z id])
> (syntax-parse #'(a b c 1 2 3)
  #:conventions (xyz-as-ids)
  [(x ... n ...) (syntax->datum #'(x ...))])
(a b c)
> (define-conventions xn-prefixes
  [#rx"^x" id]
  [#rx"^n" nat])
> (syntax-parse #'(a b c 1 2 3)
  #:conventions (xn-prefixes)
  [(x0 x ... n0 n ...) (syntax->datum #'(x0 (x ...) n0 (n ...)))])
(a (b c) 1 (2 3))

```

8.4 Library syntax classes and literal sets

8.4.1 Syntax classes

expr

Matches anything except a keyword literal (to distinguish expressions from the start of a keyword argument sequence). The term is not otherwise inspected, and no guarantee is made that the term is actually a valid expression.

identifier

boolean

str

char

keyword

number

integer

exact-integer

exact-nonnegative-integer

exact-positive-integer

Match syntax satisfying the corresponding predicates.

id

Alias for identifier.

`nat`

Alias for `exact-nonnegative-integer`.

(static *predicate description*)

Matches an identifier that is bound in the syntactic environment to static information (see [syntax-local-value](#)) satisfying the given *predicate*. If the term does not match, the *description* argument is used to describe the expected syntax.

When used outside of the dynamic extend of a macro transformer (see [syntax-transforming?](#)), matching fails.

The attribute *value* contains the value the name is bound to.

(atom-in-list *atoms description*)

Matches a syntax object whose inner datum is `eqv?` to some atom in the given list.

Use `atom-in-list` instead of a literals list when recognizing identifier based on their symbolic names rather than their bindings.

8.4.2 Literal sets

`kernel-literals`

Literal set containing the identifiers for fully-expanded code. The set contains all of the forms listed by `kernel-form-identifier-list`, plus `module`, `#:plain-module-begin`, `#:require`, and `#:provide`.

Note that the literal-set uses the names `#:plain-lambda` and `#:plain-app`, not `lambda` and `app`.

Index

- #%module-begin, 29
- atom-in-list, 56
- Attaching Documentation to Exports, 37
- attribute, 50
- attribute*, 53
- Attributes, 53
- boolean, 55
- `bound-id-table-count`, 11
- `bound-id-table-for-each`, 11
- `bound-id-table-map`, 11
- `bound-id-table-ref`, 10
- `bound-id-table-remove`, 11
- `bound-id-table-remove!`, 11
- `bound-id-table-set`, 11
- `bound-id-table-set!`, 10
- `bound-id-table?`, 10
- `bound-identifier-mapping-for-each`, 7
- `bound-identifier-mapping-get`, 6
- `bound-identifier-mapping-map`, 7
- `bound-identifier-mapping-put!`, 7
- `bound-identifier-mapping?`, 6
- `build-expand-context`, 20
- `build-struct-expand-info`, 24
- `build-struct-generation`, 23
- `build-struct-generation*`, 23
- `build-struct-names`, 22
- char, 55
- `check-module-form`, 15
- `collapse-module-path`, 18
- `collapse-module-path-index`, 18
- Computing the Free Variables of an Expression, 13
- Controlling Syntax Templates, 26
- conventions*, 54
- Deconstructing Syntax Objects, 4
- `define-conventions`, 54
- `define-literal-set`, 54
- `define-splicing-syntax-class`, 53
- `define-syntax-class`, 51
- Dictionaries for `bound-identifier=?`, 9
- Dictionaries for `free-identifier=?`, 12
- EH-pattern*, 48
- EH-pattern forms, 48
- `eval-compile-time-part-of-top-level`, 34
- `eval-compile-time-part-of-top-level/compile`, 35
- `exact-integer`, 55
- `exact-nonnegative-integer`, 55
- `exact-positive-integer`, 55
- `exn:get-module-code`, 17
- `exn:get-module-code-path`, 17
- `exn:get-module-code?`, 17
- `expand-syntax-top-level-with-compile-time-evals`, 34
- `expand-syntax-top-level-with-compile-time-evals/flatten`, 34
- `expand-top-level-with-compile-time-evals`, 34
- Expanding define-struct-like Forms, 21
- expr, 55
- Extracting Inferred Names, 20
- `flatten-begin`, 21
- Flattening begin Forms, 21
- `free-id-table-count`, 12
- `free-id-table-for-each`, 12
- `free-id-table-map`, 12
- `free-id-table-ref`, 12
- `free-id-table-remove`, 12
- `free-id-table-remove!`, 12
- `free-id-table-set`, 12
- `free-id-table-set!`, 12
- `free-id-table?`, 12
- `free-identifier-mapping-for-each`, 8
- `free-identifier-mapping-get`, 8
- `free-identifier-mapping-map`, 8
- `free-identifier-mapping-put!`, 8
- `free-identifier-mapping?`, 7
- `free-vars`, 13
- `generate-expand-context`, 20
- `generate-struct-declaration`, 24

- [get-module-code](#), 16
- Getting Module Compiled Code, 15
- H-pattern*, 47
- H-pattern variants, 47
- Hashing on [bound-identifier=?](#) and [free-identifier=?](#), 6
- id, 55
- identifier, 55
- Identifier dictionaries, 9
- [immutable-bound-id-table?](#), 10
- [immutable-free-id-table?](#), 12
- Inspecting Modules and Module Dependencies, 18
- Inspection tools, 53
- integer, 55
- [kernel-form-identifier-list](#), 6
- kernel-literals, 56
- kernel-syntax-case, 5
- kernel-syntax-case*, 5
- kernel-syntax-case*/phase, 6
- kernel-syntax-case/phase, 5
- keyword, 55
- L-pattern*, 41
- Legacy Zodiac Interface, 14
- Library syntax classes and literal sets, 55
- literal set*, 54
- Literal sets, 56
- Literal sets and Conventions, 54
- [lookup-documentation](#), 37
- Macro Transformer Helpers, 20
- [make-bound-id-table](#), 10
- [make-bound-identifier-mapping](#), 6
- [make-exn:get-module-code](#), 17
- [make-free-id-table](#), 12
- [make-free-identifier-mapping](#), 7
- [make-immutable-bound-id-table](#), 10
- [make-immutable-free-id-table](#), 12
- [make-module-identifier-mapping](#), 9
- Matching Fully-Expanded Expressions, 5
- [moddep-current-open-input-file](#), 17
- Module Reader, 29
- [module-identifier-mapping-for-](#)
 - each, 9
 - [module-identifier-mapping-get](#), 9
 - [module-identifier-mapping-map](#), 9
 - [module-identifier-mapping-put!](#), 9
 - [module-identifier-mapping?](#), 9
 - [module-or-top-identifier=?](#), 5
- Module-Processing Helpers, 15
- [mutable-bound-id-table?](#), 10
- [mutable-free-id-table?](#), 12
- nat, 56
- Non-Module Compilation And Expansion, 34
 - [normalize-definition](#), 21
- number, 55
- [parse-define-struct](#), 22
- Parsing and classifying syntax, 39
- Parsing define-like Forms, 20
- Parsing syntax, 39
- pattern, 53
- Pattern directives, 49
- [provide-and-document](#), 37
- [raise-read-eof-error](#), 29
- [raise-read-error](#), 28
- Raising [exn:fail:read](#), 28
- Reader Helpers, 28
- Reading Module Source Code, 15
- Rendering Syntax Objects with Formatting, 13
 - [replace-context](#), 13
- Replacing Lexical Context, 13
- [resolve-module-path](#), 17
- [resolve-module-path-index](#), 17
- [resolve-path-spec](#), 25
- Resolving [include-like](#) Paths, 25
- Resolving Module Paths to File Paths, 17
- S-pattern*, 41
- S-pattern variants, 41
- [show-import-tree](#), 19
- Simplifying Module Paths, 18
- static, 56
- str, 55
- [strip-context](#), 13

- [struct-declaration-info?](#), 24
- [struct:exn:get-module-code](#), 17
- [stx->list](#), 4
- [stx-car](#), 4
- [stx-cdr](#), 4
- [stx-list?](#), 4
- [stx-null?](#), 4
- [stx-pair?](#), 4
- Support for [local-expand](#), 20
- Syntax Classes, 51
- Syntax classes, 55
- Syntax Object Helpers, 4
- syntax patterns*, 40
- [syntax->string](#), 13
- [syntax-class-attributes](#), 53
- [syntax-class-parse](#), 53
- [syntax-local-infer-name](#), 20
- [syntax-parse](#), 39
- [syntax-parser](#), 40
- [syntax/boundmap](#), 6
- [syntax/context](#), 20
- [syntax/define](#), 20
- [syntax/docprovide](#), 37
- [syntax/flatten-begin](#), 21
- [syntax/free-vars](#), 13
- [syntax/id-table](#), 9
- [syntax/kerncase](#), 5
- [syntax/modcode](#), 15
- [syntax/modcollapse](#), 18
- [syntax/moddep](#), 18
- [syntax/modread](#), 15
- [syntax/modresolve](#), 17
- [syntax/module-reader](#), 29
- [syntax/name](#), 20
- [syntax/parse](#), 39
- [syntax/path-spec](#), 25
- [syntax/readerr](#), 28
- [syntax/strip-context](#), 13
- [syntax/struct](#), 21
- [syntax/stx](#), 4
- [syntax/template](#), 26
- [syntax/to-string](#), 13
- [syntax/toplevel](#), 34
- [syntax/trusted-xforms](#), 36
- [syntax/zodiac](#), 14
- [syntax/zodiac-sig](#), 14
- [syntax/zodiac-unit](#), 14
- Syntax: Meta-Programming Helpers**, 1
- [transform-template](#), 26
- Trusting Standard Recertifying Transformers, 36
- [with-module-reading-parameterization](#), 15
- [wrap-read-all](#), 33
- ~!, 50
- ~and, 50
- ~bind, 50
- ~describe, 50
- ~fail, 50
- ~once, 50
- ~optional, 50
- ~or, 50
- ~rest, 50
- ~seq, 50