

Parsing Syntax and Syntax Classes

Version 4.1.5

March 21, 2009

Warning: This library is still very volatile! Its interface and behavior are subject to frequent change. I highly recommend that you avoid creating PLaneT packages that depend on this library.

The `stxclass` library provides a framework for describing and parsing syntax. Using `stx-class`, macro writers can define new syntactic categories, specify their legal syntax, and use them to write clear, concise, and robust macros.

To load the library:

```
(require stxclass)
```

The following sections are a reference for `stxclass` features.

1 Parsing Syntax

This section describes `stxclass`'s facilities for parsing syntax.

```
(syntax-parse stx-expr maybe-literals clause ...)

maybe-literals =
  | #:literals (literal ...)

  literal = id
  | (internal-id external-id)

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

Evaluates `stx-expr`, which should produce a syntax object, and matches it against the patterns in order. If some pattern matches, its pattern variables 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 syntax error indicates the first specific subterm for which no pattern matches.

A literal in the literals list has two components: the identifier used within the pattern to signify the positions to be matched, and the identifier expected to occur in those positions. If the single-identifier form is used, the same identifier is used for both purposes.

```
(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 patterns accepted by `syntax-parse` and `syntax-parser` follows:

```
syntax-pattern = pvar-id
  | pvar-id:syntax-class-id
  | literal-id
  | atomic-datum
  | (syntax-pattern . syntax-pattern)
  | (ellipsis-head-pattern ... . syntax-pattern)
  | (~and maybe-description syntax-pattern ...)

ellipsis-head-pattern = (~or head ...+)
  | syntax-pattern
```

```
maybe-description =  
    | #:description string  
  
pvar-id = _  
    | id
```

Here are the variants of `syntax-pattern`:

pvar-id

Matches anything. The pattern variable is bound to the matched subterm, unless the pattern variable is the wildcard (`_`), in which case no binding occurs.

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 typically bound to the matched subterm, but the syntax class can substitute a transformed subterm instead.

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

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

atomic-datum

The empty list, numbers, strings, booleans, and keywords match as literals.

(syntax-pattern . syntax-pattern)

Matches a syntax pair whose head matches the first pattern and whose tail matches the second.

(ellipsis-head-pattern syntax-pattern)

Matches a sequence of the first pattern ending in a tail matching the second pattern.

That is, the sequence pattern matches either the second pattern (which need not be a list) or a pair whose head matches the first pattern and whose tail recursively matches the whole sequence pattern.

The head pattern can be either an ordinary pattern or an or/sequence-pattern:

```
(~or head ...+)
```

```
head = (syntax-pattern ...+) head-directive ...
```

```
head-directive = #:min min-reps  
                | #:max max-reps  
                | #:mand
```

If the head is an or/sequence-pattern (introduced by `~or`), then the whole sequence pattern matches any combination of the head sequences followed by a tail matching the final pattern.

```
#:min min-reps
```

Requires at least `min-reps` occurrences of the preceding head to match. `min-reps` must be a literal exact nonnegative integer.

```
#:max max-reps
```

Requires that no more than `max-reps` occurrences of the preceding head to match. `max-reps` must be a literal exact nonnegative integer, and it must be greater than or equal to `min-reps`.

```
#:mand
```

Requires that the preceding head occur exactly once. Pattern variables in the preceding head are not bound at a higher ellipsis nesting depth.

```
(~and maybe-description syntax-pattern ...)
```

```
maybe-description =  
                    | #:description string
```

Matches any syntax that matches all of the included patterns.

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
                    | #:when 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 *exprs* are evaluated outside the scope of the pattern variable bindings.

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

Evaluates the *expr* in the context of all previous pattern variable bindings and matches it against the pattern. If the match succeeds, the new pattern variables are added to the 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.

```
#:when expr
```

Evaluates the *expr* in the context of all previous pattern variable bindings. If it produces a false value, the matching process backtracks as described above; otherwise, it continues.

`~and`

Keyword recognized by `syntax-parse` etc as notation for and-patterns.

`~or`

Keyword recognized by `syntax-parse` etc as notation for or/sequence-patterns (within se-

quences). It may not be used as an expression.

(`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.

2 Syntax Classes

Syntax classes provide an abstraction mechanism for the specification of syntax. Basic syntax classes include `identifier` and `keyword`. More generally, a programmer can define a “basic” syntax from an arbitrary predicate, although syntax classes thus defined lose some of the benefits of declarative specification of syntactic structure.

Programmers can also 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, includes) 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-body)
(define-syntax-class (name-id arg-id ...) stxclass-option ...
  stxclass-body)

stxclass-options = #:attributes (attr-arity-decl ...)
                  | #:description description
                  | #:transparent
                  | #:literals (literal-entry ...)

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

stxclass-body = (pattern syntax-pattern stxclass-pattern-directive ...) ...+
                 | (basic-syntax-class parser-expr)
```

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 either one `basic-syntax-class` clause or a non-empty sequence of pattern clauses.

```
#: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 computed using attribute inference.

`#:description description`

The *description* argument is an expression (with the syntax-class's parameters in scope) 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 generated.

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)`

Declares the literal identifiers for the syntax class's main patterns (immediately within pattern variants) and `#:with` clauses. The literals list does not affect patterns that occur within subexpressions inside the syntax class (for example, the condition of a `#:when` clause or the right-hand side of a `#:with` binding).

A literal can have separate internal and external names, as described for `syntax-parse`.

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

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

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

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

The name of an attribute is the symbolic name of the pattern variable, except when the name is explicitly given via a `#:rename` clause.

`#:rename internal-id external-id`

Exports the pattern variable binding named by *internal-id* as the attribute named *external-id*.

(`basic-syntax-class parser-expr`)

The `parser-expr` must evaluate to a procedure. This procedure is used to parse or reject syntax objects. The arguments to the parser procedure consist of the syntax object to parse followed by the syntax-class parameterization arguments (the parameter names given at the `define-syntax-class` level are not bound within the `parser-expr`). To indicate success, the parser should return a list of attribute values, one for each attribute listed. (For example, a parser for a syntax class that defines no attributes returns the empty list when it succeeds.) To indicate failure, the parser procedure should return `#f`.

The parser procedure should avoid side-effects, as they interfere with the parsing process's backtracking and error reporting.

TODO: Add support for better error reporting within basic syntax class.

`pattern`

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

`basic-syntax-class`

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

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 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`.

If the attributes are not declared explicitly, they are computed via *attribute inference*. For “basic” syntax classes, the inferred attribute list is always empty. For compound syntax classes, each `pattern` form is analyzed to determine its candidate attributes. The attributes of the syntax class are the attributes common to all of the variants (that is, the intersection of the candidate attributes). An attribute must have the same ellipsis-depth in each of the variants; otherwise, an error is raised.

The candidate attributes of a `pattern` variant are the pattern variables bound by the variant's

pattern (including nested attributes contributed by their associated syntax classes) together with the pattern variables (and nested attributes) from `#:with` clauses.

For the purpose of attribute inference, recursive references to the same syntax class and forward references to syntax classes not yet defined do not contribute any nested attributes. This avoids various problems in computing attributes, including infinitely nested attributes.

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.

3 Library syntax classes

`expr`

Matches anything except a keyword literal (to distinguish expressions from the start of a keyword argument sequence). Does not expand or otherwise inspect the term.

`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`.

The following syntax classes mirror parts of the macro API. They may only be used during transformation (when `syntax-transforming?` returns true). Otherwise they may raise an error.

`static`

Matches identifiers that are bound in the syntactic environment to static information (see `syntax-local-value`). Attribute `value` contains the value the name is bound to.

`(static-of description predicate)`

Refines `static`: matches identifiers that are bound in the syntactic environment to static information satisfying the given `predicate`. Attribute `value` contains the value the name is bound to. The `description` argument is used for error reporting.

4 Utilities

The `stxclass` collection includes several utility modules. They are documented individually below.

As a shortcut, the `stxclass/util` module provides all of the contents of the separate utility modules:

```
(require stxclass/util)
```

The contents of the utility modules are not provided by the main `stxclass` module.

4.1 Error reporting

```
(require stxclass/util/error)
```

The `scheme/base` and `scheme` languages provide the `raise-syntax-error` procedure for reporting syntax errors. Using `raise-syntax-error` effectively requires passing around either a symbol indicating the special form that signals the error or else a “contextual” syntax object from which the special form’s name can be extracted. This library helps manage the contextual syntax for reporting errors.

```
(current-syntax-context) → (or/c syntax? false/c)  
(current-syntax-context stx) → void?  
  stx : (or/c syntax? false/c)
```

The current contextual syntax object, defaulting to `#f`. It determines the special form name that prefixes syntax errors created by `wrong-syntax`, as follows:

If it is a syntax object with a `'report-error-as` syntax property whose value is a symbol, then that symbol is used as the special form name. Otherwise, the same rules apply as in `raise-syntax-error`.

```
(wrong-syntax stx format-string v ...) → any  
  stx : syntax?  
  format-string : string?  
  v : any/c
```

Raises a syntax error using the result of `(current-syntax-context)` as the “major” syntax object and the provided `stx` as the specific syntax object. (The latter, `stx`, is usually the one highlighted by DrScheme.) The error message is constructed using the format string and arguments, and it is prefixed with the special form name as described under `current-syntax-context`.

A macro using this system might set the syntax context at the very beginning of its transformation as follows:

```
(define-syntax (my-macro stx)
  (parameterize ((current-syntax-context stx))
    (syntax-case stx ()
      __)))
```

Then any calls to `wrong-syntax` during the macro's transformation will refer to `my-macro` (more precisely, the name that referred to `my-macro` where the macro was used, which may be different due to renaming, prefixing, etc).

A macro that expands into a helper macro can insert its own name into syntax errors raised by the helper macro by installing a `'report-error-as` syntax property on the helper macro expression. For example:

```
(define-syntax (public-macro stx)
  (syntax-case stx ()
    [(public-macro stuff)
     (syntax-property
      (syntax/loc stx (my-macro stuff other-internal-stuff))
      'report-error-as
      (syntax-e #'public-macro))]))
```

4.2 Miscellaneous utilities

```
(require stxclass/util/misc)
```

```
(define-pattern-variable id expr)
```

Evaluates `expr` and binds it to `id` as a pattern variable, so `id` can be used in subsequent syntax patterns.

```
(with-temporaries (temp-id ...) . body)
```

Evaluates `body` with each `temp-id` bound as a pattern variable to a freshly generated identifier.

For example, the following are equivalent:

```
(with-temporaries (x) #'(lambda (x) x))
(with-syntax ([(x) (generate-temporaries '(x))])
  #'(lambda (x) x))
```

`(generate-temporary) → identifier?`

Generates one fresh identifier. Singular form of `generate-temporaries`.

`(generate-n-temporaries n) → (listof identifier?)`
`n : exact-nonnegative-integer?`

Generates a list of n fresh identifiers.

`(with-catching-disappeared-uses body-expr)`

Evaluates the `body-expr`, catching identifiers looked up using `syntax-local-value/catch`. Returns two values: the result of `body-expr` and the list of caught identifiers.

`(with-disappeared-uses stx-expr)`

Evaluates the `stx-expr`, catching identifiers looked up using `syntax-local-value/catch`. Adds the caught identifiers to the `'disappeared-uses` syntax property of the resulting syntax object.

`(syntax-local-value/catch id predicate) → any/c`
`id : identifier?`
`predicate : (-> any/c boolean?)`

Looks up `id` in the syntactic environment (as `syntax-local-value`). If the lookup succeeds and returns a value satisfying the predicate, the value is returned and `id` is recorded (“caught”) as a disappeared use. If the lookup fails or if the value does not satisfy the predicate, `#f` is returned and the identifier is not recorded as a disappeared use.

`(chunk-kw-seq stx table [context])`
`→ (listof (cons/c keyword? (cons/c (syntax/c keyword?) list?)))`
`syntax?`
`stx : syntax?`
`table : (listof (cons/c keyword?`
`(listof (-> syntax? any))))`
`context : (or/c syntax? false/c) = #f`

Parses a syntax list into keyword-argument “chunks” and a syntax list tail (the remainder of the syntax list). The syntax of the keyword arguments is specified by `table`, an association list mapping keywords to lists of checker procedures. The length of the checker list is the number of “arguments” expected to follow the keyword, and each checker procedure is applied to the corresponding argument. The result of the checker procedure is entered into the

chunk for that keyword sequence. The same keyword can appear multiple times in the result list.

The *context* is used to report errors.

```
(chunk-kw-seq/no-dups stx table [context])  
→ (listof (cons/c keyword? (cons/c (syntax/c keyword?) list?)))  
  syntax?  
  stx : syntax?  
  table : (listof (cons/c keyword?  
                  (listof (-> syntax? any))))  
  context : (or/c syntax? false/c) = #f
```

Like `chunk-kw-seq` filtered by `reject-duplicate-chunks`.

The *context* is used to report errors.

```
(reject-duplicate-chunks chunks) → void?  
  chunks : (listof (cons/c keyword? (cons/c (syntax/c keyword?) list?)))
```

Raises a syntax error if it encounters the same keyword more than once in the *chunks* list.

The *context* is used to report errors.

4.3 Structs

```
(require stxclass/util/struct)
```

```
(make struct-id v ...)
```

Constructs an instance of *struct-id*, which must be defined as a struct name. If *struct-id* has a different number of fields than the number of *v* values provided, `make` raises a compile-time error.