## 4.2.3 Documenting Forms, Functions, Structure Types, and Values

```
(defproc options prototype                                    syntax
        result-contract-expr-datum
        maybe-value
        pre-flow ...)

    prototype = (id arg-spec ...)
              | (prototype arg-spec ...)

     arg-spec = (arg-id contract-expr-datum)
              | (arg-id contract-expr-datum default-expr)
              | (keyword arg-id contract-expr-datum)
              | (keyword arg-id contract-expr-datum default-expr)
              | ellipses
              | ellipses+

      options = maybe-kind maybe-link maybe-id

   maybe-kind =
              | #:kind kind-content-expr

   maybe-link =
              | #:link-target? link-target?-expr

     maybe-id =
              | #:id [src-id dest-id-expr]

  maybe-value =
              | #:value value-expr-datum

     ellipses = ...

    ellipses+ = ...+
```

Produces a sequence of flow elements (encapsulated in a `splice`) to document a procedure named *id*. Nesting *prototype*s corresponds to a curried function, as in `define`. Unless *link-target?-expr* is specified and produces `#f`, the *id* is indexed, and it also registered so that `racket`-typeset uses of the identifier (with the same for-label binding) are hyperlinked to this documentation.

Examples:

```
@defproc[(make-sandwich [ingredients (listof ingredient?)])
         sandwich?]{
  Returns a sandwich given the right ingredients.
}

@defproc[#:kind "sandwich-maker"
         (make-reuben [ingredient sauerkraut?] ...
                      [#:veggie? veggie? any/c #f])
         sandwich?]{
  Produces a reuben given some number of @racket[ingredient]s.

  If @racket[veggie?] is @racket[#f], produces a standard
  reuben with corned beef. Otherwise, produces a vegetable
  reuben.
}
```

Renders like:

```
(make-sandwich ingredients) → sandwich?                procedure
  ingredients : (listof ingredient?)
```

Returns a sandwich given the right ingredients.

```
(make-reuben  ingredient                          sandwich-maker
              ...
             [#:veggie? veggie?]) → sandwich?
  ingredient : sauerkraut?
  veggie? : any/c = #f
```

Produces a reuben given some number of *ingredient*s.

If *veggie?* is `#f`, produces a standard reuben with corned beef. Otherwise, produces a vegetable reuben.

When *id* is indexed and registered, a `defmodule` or `declare-exporting` form (or one of the variants) in an enclosing section determines the *id* binding that is being defined. The *id* should also have a for-label binding (as introduced by `(require (for-label ....))`) that matches the definition binding; otherwise, the defined *id* will not typeset correctly within the definition.

Each *arg-spec* must have one of the following forms:

```
(arg-id contract-expr-datum)
```

An argument whose contract is specified by *contract-expr-datum* which is typeset via `racketblock0`.

```
(arg-id contract-expr-datum default-expr)
```

Like the previous case, but with a default value. All arguments with a default value must be grouped together, but they can be in the middle of required arguments.

```
(keyword arg-id contract-expr-datum)
```

Like the first case, but for a keyword-based argument.

```
(keyword arg-id contract-expr-datum default-expr)
```

Like the previous case, but with a default value.

```
...
```

Any number of the preceding argument. This form is normally used at the end, but keyword-based arguments can sensibly appear afterward. See also the documentation for `append` for a use of `...` before the last argument.

```
...+
```

One or more of the preceding argument (normally at the end, like `...`).

The *`result-contract-expr-datum`* is typeset via `racketblock0`, and it represents a contract on the procedure's result.

The decoded *`pre-flow`* documents the procedure. In this description, references to *`arg-id`*s using `racket`, `racketblock`, etc. are typeset as procedure arguments.

The typesetting of all information before the *`pre-flow`*s ignores the source layout, except that the local formatting is preserved for contracts and default-values expressions. The information is formatted to fit (if possible) in the number of characters specified by the `current-display-width` parameter.

An optional `#:kind` specification chooses the decorative label, which defaults to `"procedure"`. A `#f` result for *`kind-content-expr`* uses the default, otherwise *`kind-content-expr`* should produce content in the sense of `content?`. An alternate label should be all lowercase.

If `#:id [src-id dest-id-expr]` is supplied, then *`src-id`* is the identifier as it appears in the *`prototype`* (to be replaced by a defining instance), and *`dest-id-expr`* produces the identifier to be documented in place of *`src-id`*. This split between *`src-id`* and *`dest-id-expr`* roles is useful for functional abstraction of `defproc`.

If `#:value value-expr-datum` is given, *`value-expr-datum`* is typeset using `racketblock0` and included in the documentation. As a service to readers, please use `#:value` to document only simple, short functions.

```
(defproc* options                                          syntax
          ([prototype
            result-contract-expr-datum
            maybe-value] ...+)
          pre-flow ...)
```

Like `defproc`, but for multiple cases with the same `id`. Multiple distinct `id`s can also be defined by a single `defproc*`, for the case that it's best to document a related group of procedures at once (but multiple `defproc`s grouped by `deftogether` also works for that case).

When an `id` has multiple calling cases, either they must be defined with a single `defproc*`, so that a single definition point exists for the `id`, or else all but one definition should use `#:link-target? #f`.

Examples:

```
@defproc*[([(make-pb&j) sandwich?]
           [(make-pb&j [jelly jelly?]) sandwich?])]{
  Returns a peanut butter and jelly sandwich. If @racket[jelly]
  is provided, then it is used instead of the standard (grape)
  jelly.
}
```

Renders like:

```
(make-pb&j) → sandwich?                                procedure
(make-pb&j jelly) → sandwich?
  jelly : jelly?
```

Returns a peanut butter and jelly sandwich. If *jelly* is provided, then it is used instead of the standard (grape) jelly.

```
(defform options form-datum                                       syntax
  maybe-grammar maybe-contracts
  pre-flow ...)

         options  =  maybe-kind maybe-link maybe-id maybe-literals

      maybe-kind  =
                  |  #:kind kind-content-expr

      maybe-link  =
                  |  #:link-target? link-target?-expr

        maybe-id  =
                  |  #:id id
                  |  #:id [id id-expr]

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

   maybe-grammar  =
                  |  #:grammar ([nonterm-id clause-datum ...+] ...)

 maybe-contracts  =
                  |  #:contracts ([subform-datum contract-expr-datum]
                                  ...)
```

Produces a sequence of flow elements (encapsulated in a `splice`) to document a syntatic form named by *`id`* (or the result of *`id-expr`*) whose syntax is described by *`form-datum`*. If no `#:id` is used to specify *`id`*, then *`form-datum`* must have the form `(id . datum)`.

If `#:kind kind-content-expr` is supplied, it is used in the same way as for `defproc`, but the default kind is `"syntax"`.

If `#:id [id id-expr]` is supplied, then *`id`* is the identifier as it appears in the *`form-datum`* (to be replaced by a defining instance), and *`id-expr`* produces the identifier to be documented. This split between *`id`* and *`id-expr`* roles is useful for functional abstraction of `defform`.

Unless *`link-target?-expr`* is specified and produces `#f`, the *`id`* (or result of *`id-expr`*) is indexed, and it is also registered so that `racket`-typeset uses of the identifier (with the same for-label binding) are hyperlinked to this documentation. The `defmodule` or `declare-exporting` requirements, as well as the binding requirements for *`id`* (or result of *`id-expr`*), are the same as for `defproc`.

The decoded *`pre-flow`* documents the form. In this description, a reference to any identifier in *`form-datum`* via `racket`, `racketblock`, etc. is typeset as a sub-form non-terminal. If `#:literals` clause is provided, however, instances of the *`literal-id`*s are typeset normally (i.e., as determined by the enclosing context).

If a `#:grammar` clause is provided, it includes an auxiliary grammar of non-terminals shown with the *`id`* form. Each *`nonterm-id`* is specified as being any of the corresponding *`clause-datum`*s.

If a `#:contracts` clause is provided, each *`subform-datum`* (typically an identifier that serves as a meta-variable in *`form-datum`* or *`clause-datum`*) is shown as producing a value that must satisfy the contract described by *`contract-expr-datum`*. Use `#:contracts` only to specify constraints on a *value* produced by an expression; for constraints on the *syntax* of a *`subform-datum`*, use grammar notation instead, possibly through an auxiliary grammar specified with `#:grammar`.

The typesetting of *`form-datum`*, *`clause-datum`*, *`subform-datum`*, and *`contract-expr-datum`* preserves the source layout, like `racketblock`.

Examples:

```
@defform[(sandwich-promise sandwich-expr)
         #:contracts ([sandwich-expr sandwich?])]{
  Returns a promise to construct a sandwich. When forced, the promise
  will produce the result of @racket[sandwich-expr].
}

@defform[#:literals (sandwich mixins)
         (sandwich-promise* [sandwich sandwich-expr]
                            [mixins ingredient-expr ...])
         #:contracts ([sandwich-expr sandwich?]
                      [ingredient-expr ingredient?])]{
  Returns a promise to construct a sandwich. When forced, the promise
  will produce the result of @racket[sandwich-expr]. Each result of
```

```
    the @racket[ingredient-expr]s will be mixed into the resulting
    sandwich.
  }

  @defform[(sandwich-factory maybe-name factory-component ...)
           #:grammar
           [(maybe-name (code:line)
                        name)
            (factory-component (code:line #:protein protein-expr)
                               [vegetable vegetable-expr])]]{
    Constructs a sandwich factory. If @racket[maybe-name] is provided,
    the factory will be named. Each of the @racket[factory-component]
    clauses adds an additional ingredient to the sandwich pipeline.
  }
```

Renders like:

```
(sandwich-promise sandwich-expr)                        syntax

  sandwich-expr : sandwich?
```

Returns a promise to construct a sandwich. When forced, the promise will produce the result of `sandwich-expr`.

```
(sandwich-promise* [sandwich sandwich-expr]             syntax
                   [mixins ingredient-expr ...])

  sandwich-expr : sandwich?
  ingredient-expr : ingredient?
```

Returns a promise to construct a sandwich. When forced, the promise will produce the result of `sandwich-expr`. Each result of the `ingredient-expr`s will be mixed into the resulting sandwich.

```
(sandwich-factory maybe-name factory-component ...)     syntax

         maybe-name  =
                     |  name

  factory-component  =  #:protein protein-expr
                     |  [vegetable vegetable-expr]
```

Constructs a sandwich factory. If `maybe-name` is provided, the factory will be named. Each of the `factory-component` clauses adds an additional ingredient to the sandwich pipeline.

```
(defform* options [form-datum ...+]                            syntax
  maybe-grammar maybe-contracts
  pre-flow ...)
```

Like `defform`, but for multiple forms using the same *id*.

Examples:

```
@defform*[((call-with-current-sandwich expr)
           (call-with-current-sandwich expr sandwich-handler-expr))]{
  Runs @racket[expr] and passes it the value of the current
  sandwich. If @racket[sandwich-handler-expr] is provided, its result
  is invoked when the current sandwich is eaten.
}
```

Renders like:

```
(call-with-current-sandwich expr)                          syntax
(call-with-current-sandwich expr sandwich-handler-expr)
```

Runs *expr* and passes it the value of the current sandwich. If *sandwich-handler-expr* is provided, its result is invoked when the current sandwich is eaten.

```
(defform/none maybe-kind maybe-literal form-datum                   syntax
  maybe-grammar maybe-contracts
  pre-flow ...)
```

Like `defform` with `#:link-target? #f`.

```
(defidform maybe-kind maybe-link id pre-flow ...)                   syntax
```

Like `defform`, but with a plain *id* as the form.

```
(defidform/inline id)                                               syntax
(defidform/inline (unsyntax id-expr))
```

Like `defidform`, but *id* (or the result of *id-expr*, analogous to `defform`) is typeset as an inline element. Use this form sparingly, because the typeset form does not stand out to the reader as a specification of *id*.

```
(defsubform options form-datum                                      syntax
   maybe-grammar maybe-contracts
   pre-flow ...)
(defsubform* options [form-datum ...+]                              syntax
  maybe-grammar maybe-contracts
  pre-flow ...)
```

Like `defform` and `defform*`, but with indenting on the left for both the specification and the *pre-flow*s.

```
(specform maybe-literals datum maybe-grammar maybe-contracts        syntax
  pre-flow ...)
```

Like `defform` with `#:link-target? #f`, but with indenting on the left for both the specification and the *`pre-flow`*s.

```
(specsubform maybe-literals datum maybe-grammar maybe-contracts     syntax
  pre-flow ...)
```

Similar to `defform` with `#:link-target? #f`, but without the initial identifier as an implicit literal, and the table and flow are typeset indented. This form is intended for use when refining the syntax of a non-terminal used in a `defform` or other `specsubform`. For example, it is used in the documentation for `defproc` in the itemization of possible shapes for *`arg-spec`*.

The *`pre-flow`*s list is parsed as a flow that documents the procedure. In this description, a reference to any identifier in *`datum`* is typeset as a sub-form non-terminal.

```
(specspecsubform maybe-literals datum maybe-grammar maybe-contracts syntax
  pre-flow ...)
```

Like `specsubform`, but indented an extra level. Since using `specsubform` within the body of `specsubform` already nests indentation, `specspecsubform` is for extra indentation without nesting a description.

```
(defform/subs options form-datum                                    syntax
  ([nonterm-id clause-datum ...+] ...)
  maybe-contracts
  pre-flow ...)
(defform*/subs options [form-datum ...+]                            syntax
  ([nonterm-id clause-datum ...+] ...)
  maybe-contracts
  pre-flow ...)
(specform/subs maybe-literals datum                                 syntax
  ([nonterm-id clause-datum ...+] ...)
  maybe-contracts
  pre-flow ...)
(specsubform/subs maybe-literals datum                              syntax
  ([nonterm-id clause-datum ...+] ...)
  maybe-contracts
  pre-flow ...)
(specspecsubform/subs maybe-literals datum                          syntax
 ([nonterm-id clause-datum ...+] ...)
 maybe-contracts
 pre-flow ...)
```

Like `defform`, `defform*`, `specform`, `specsubform`, and `specspecsubform`, respectively, but the auxiliary grammar is mandatory and the `#:grammar` keyword is omitted.

Examples:

```
@defform/subs[(sandwich-factory maybe-name factory-component ...)
              [(maybe-name (code:line)
                           name)
               (factory-component (code:line #:protein protein-expr)
                                  [vegetable vegetable-expr])]]{
  Constructs a sandwich factory. If @racket[maybe-name] is provided,
  the factory will be named. Each of the @racket[factory-component]
  clauses adds an additional ingredient to the sandwich pipeline.
}
```

Renders like:

```
(sandwich-factory maybe-name factory-component ...)      syntax

         maybe-name =
                    | name

  factory-component = #:protein protein-expr
                    | [vegetable vegetable-expr]
```

Constructs a sandwich factory. If *maybe-name* is provided, the factory will be named. Each of the *factory-component* clauses adds an additional ingredient to the sandwich pipeline.

```
(defparam maybe-link id arg-id                                  syntax
  contract-expr-datum
  maybe-value
  pre-flow ...)
```

Like `defproc`, but for a parameter. The *contract-expr-datum* serves as both the result contract on the parameter and the contract on values supplied for the parameter. The *arg-id* refers to the parameter argument in the latter case.

Examples:

```
@defparam[current-sandwich sandwich sandwich?
          #:value empty-sandwich]{
  A parameter that defines the current sandwich for operations that
  involve eating a sandwich. Default value is the empty sandwich.
}
```

Renders like:

```
(current-sandwich) → sandwich?                          parameter
(current-sandwich sandwich) → void?
  sandwich : sandwich?
 = empty-sandwich
```

A parameter that defines the current sandwich for operations that involve eating a sandwich. Default value is the empty sandwich.

```
(defparam* maybe-link id arg-id                                syntax
  in-contract-expr-datum out-contract-expr-datum
  maybe-value
  pre-flow ...)
```

Like `defparam`, but with separate contracts for when the parameter is being set versus when it is being retrieved (for the case that a parameter guard coerces values matching a more flexible contract to a more restrictive one; `current-directory` is an example).

```
(defboolparam maybe-link id arg-id                             syntax
  maybe-value
  pre-flow ...)
```

Like `defparam`, but the contract on a parameter argument is `any/c`, and the contract on the parameter result is `boolean?`.

```
(defthing options id contract-expr-datum maybe-value           syntax
  pre-flow ...)

     options = maybe-kind maybe-link maybe-id

  maybe-kind =
             | #:kind kind-content-expr

  maybe-link =
             | #:link-target? link-target?-expr

    maybe-id =
             | #:id id-expr

 maybe-value =
             | #:value value-expr-datum
```

Like `defproc`, but for a non-procedure binding.

If `#:kind` *`kind-content-expr`* is supplied, it is used in the same way as for `defproc`, but the default kind is `"value"`.

If `#:id` *`id-expr`* is supplied, then the result of *`id-expr`* is used in place of *`id`*.

If `#:value` *`value-expr-datum`* is given, *`value-expr-datum`* is typeset using `racketblock0` and included in the documentation. Wide values are put on a separate line.

Examples:

```
@defthing[moldy-sandwich sandwich?]{
  Don't eat this. Provided for backwards compatibility.
}

@defthing[empty-sandwich sandwich? #:value (make-sandwich empty)]{
  The empty sandwich.
}
```

Renders like:

```
moldy-sandwich : sandwich?                                value
```

Don’t eat this. Provided for backwards compatibility.

```
empty-sandwich : sandwich? = (make-sandwich empty)        value
```

The empty sandwich.

```
(defthing* options ([id contract-expr-datum maybe-value] ...+)         syntax
  pre-flow ...)
```

Like `defthing`, but for multiple non-procedure bindings. Unlike `defthing`, *`id-expr`* is not supported.

Examples:

```
@defthing*[([moldy-sandwich sandwich?]
            [empty-sandwich sandwich?])]{
  Predefined sandwiches.
}
```

Renders like:

```
moldy-sandwich : sandwich?                                value
empty-sandwich : sandwich?
```

Predefined sandwiches.

```
(defstruct* maybe-link struct-name ([field-name contract-expr-datum] ...) syntax
  maybe-mutable maybe-non-opaque maybe-constructor
  pre-flow ...)
(defstruct maybe-link struct-name ([field-name contract-expr-datum] ...)  syntax
  maybe-mutable maybe-non-opaque maybe-constructor
  pre-flow ...)

       maybe-link  =
                   |  #:link-target? link-target?-expr

      struct-name  =  id
                   |  (id super-id)

    maybe-mutable  =
                   |  #:mutable

 maybe-non-opaque  =
                   | #:prefab
                   | #:transparent
                   | #:inspector #f

maybe-constructor =
                   | #:constructor-name constructor-id
                   | #:extra-constructor-name constructor-id
                   | #:omit-constructor
```

Similar to `defform` or `defproc`, but for a structure definition. The `defstruct*` form corresponds to `struct`, while `defstruct` corresponds to `define-struct`.

Examples:

An example using `defstruct`:

```
@defstruct[sandwich ([protein ingredient?] [sauce ingredient?])]{
  A structure type for sandwiches. Sandwiches are a pan-human foodstuff
  composed of a partially-enclosing bread material and various
  ingredients.
}
```

Renders like:

```
(struct sandwich (protein sauce)                         struct
    #:extra-constructor-name make-sandwich)
  protein : ingredient?
  sauce : ingredient?
```

A structure type for sandwiches. Sandwiches are a pan-human foodstuff composed of a partially-enclosing bread material and various ingredients.

Additionally, an example using `defstruct*`:

```
@defstruct*[burrito ([salsa ingredient?] [tortilla ingredient?])]{
  A structure type for burritos. Burritos are a pan-human foodstuff
  composed of a @emph{fully}-encolosed bread material and various
  ingredients.
}
```

Renders like:

```
(struct burrito (salsa tortilla))                        struct
  salsa : ingredient?
  tortilla : ingredient?
```

A structure type for burritos. Burritos are a pan-human foodstuff composed of a *fully*-encolosed bread material and various ingredients.

```
(deftogether [def-expr ...+] pre-flow ...)                      syntax
```

Combines the definitions created by the *def-expr*s into a single definition box. Each *def-expr* should produce a definition point via `defproc`, `defform`, etc. Each *def-expr* should have an empty *pre-flow*; the decoded *pre-flow* sequence for the `deftogether` form documents the collected bindings.

Examples:

```
@deftogether[(@defthing[test-sandwich-1 sandwich?]
              @defthing[test-sandwich-2 sandwich?])]{
  Two high-quality sandwiches. These are provided for convenience
  in writing test cases
}
```

Renders like:

```
test-sandwich-1 : sandwich?                                value
test-sandwich-2 : sandwich?                                value
```

Two high-quality sandwiches. These are provided for convenience in writing test cases

```
(racketgrammar maybe-literals id clause-datum ...+)              syntax

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

Creates a table to define the grammar of *id*. Each identifier mentioned in a *clause-datum* is typeset as a non-terminal, except for the identifiers listed as *literal-id*s, which are typeset as with `racket`.

```
(racketgrammar* maybe-literals [id clause-datum ...+] ...)       syntax
```

Like `racketgrammar`, but for typesetting multiple productions at once, aligned around the `=` and `|`.

```
(defidentifier  id                                            procedure
               [#:form? form?
                #:index? index?
                #:show-libs? show-libs?]) → element?
  id : identifier?
  form? : boolean? = #f
  index? : boolean? = #t
  show-libs? : boolean? = #t
```

Typesets *id* as a Racket identifier, and also establishes the identifier as the definition of a binding in the same way as `defproc`, `defform`, etc. As always, the library that provides the identifier must be declared via `defmodule` or `declare-exporting` for an enclosing section.

If `form?` is a true value, then the identifier is documented as a syntactic form, so that uses of the identifier (normally including `id` itself) are typeset as a syntactic form.

If `index?` is a true value, then the identifier is registered in the index.

If `show-libs?` is a true value, then the identifier's defining module may be exposed in the typeset form (e.g., when viewing HTML and the mouse hovers over the identifier).

---

```
(schemegrammar maybe-literals id clause-datum ...+)                    syntax
(schemegrammar* maybe-literals [id clause-datum ...+] ...)             syntax
```

Compatibility aliases for `racketgrammar` and `racketgrammar*`.

---

```
(current-display-width) → exact-nonnegative-integer?                parameter
(current-display-width w) → void?
  w : exact-nonnegative-integer?
```

Specifies the target maximum width in characters for the output of `defproc` and `defstruct`.