DrRacket Tools

Version 9.4.0.4

Robert Bruce Findler

November 5, 2026

This manual describes portions of DrRacket's functionality that are exposed via Racket APIs to be used with other editors.

Contents

1 Accessing Check Syntax Programmatically **3**

2 Module Browser **22**

3 Module Path Selection **23**

3.1 GUI Module Path Selection 23

3.2 Module Path Selection Completion Computation 23

1 Accessing Check Syntax Programmatically

```
(require drracket/check-syntax)
    package: drracket-tool-text-lib
```

```
(show-content file-or-stx
              [#:fully-expanded? fully-expanded?
               #:namespace namespace])
 → (listof vector?)
  file-or-stx : (or/c path-string?
                      (and/c syntax?
                             (λ (x) (path-string? (syntax-source x)))))
  fully-expanded? : boolean? = #f
  namespace : (or/c #f namespace?) = #f
```

This procedure provides a simplified interface to the rest of the library, as shown in the example below. The list it returns has one vector for each call that would be made to the object in `current-annotations`. Each vector's first element is a symbol naming a method in `syncheck-annotations<%>` and the other elements of the vector are the arguments passed to the method. (Note that this procedure does not account for the callback procedures present in `syncheck:add-arrow/name-dup/pxpy`.)

The *file-or-stx* argument gives the input program and *fully-expanded?* indicates if the *file-or-stx* argument has already been fully expanded (it is ignored if *file-or-stx* is not syntax). The *namespace* argument is installed as the `current-namespace` or, if *namespace* is `#f`, then a new namespace is created, using `(make-base-namespace)`.

See `annotations-mixin` for some example code to use the other parts of this library.

Note that the paths in the example below have been replaced via `make-paths-be-module-paths` in order to make the results be platform independent.

```
> (define (make-paths-be-module-paths x)
    (let loop ([x x])
      (cond
        [(pair? x) (cons (loop (car x)) (loop (cdr x)))]
        [(vector? x) (for/vector ([x (in-vector x)])
                       (loop x))]
        [(path? x) (path->relative-string/library x)]
        [else x])))
> (let ([example-module
         '(module m racket (λ (x) x))])
    (make-paths-be-module-paths
     (show-content
      (read-syntax
       (build-path (current-directory) "dummy-file.rkt")
```

```
      (open-input-string (format "~s" example-module))))))
'(#(syncheck:add-require-open-menu 10 16 "<collects>/racket/main.rkt")
  #(syncheck:add-tail-arrow 17 25)
  #(syncheck:add-text-type 18 20 document-identifier)
  #(syncheck:add-docs-menu
    18
    20
    λ
    "View documentation for ‘λ’ from racket/base, racket"
    "<doc>/reference/lambda.html"
    (form ((lib "racket/private/base.rkt") λ))
    "(form._((lib._racket/private/base..rkt)._~ce~bb))")
  #(syncheck:add-jump-to-definition/phase-level+space
    18
    20
    new-lambda
    "<collects>/racket/private/kw.rkt"
    ()
    0)
  #(syncheck:add-mouse-over-status 18 20 "imported from racket")
  #(syncheck:add-arrow/name-dup/pxpy
    10
    16
    0.5
    0.5
    18
    20
    0.5
    0.5
    #t
    0
    module-lang
    #<procedure:name-dup?>)
  #(syncheck:add-text-type 18 20 document-identifier)
  #(syncheck:add-docs-menu
    18
    20
    λ
    "View documentation for ‘λ’ from racket/base, racket"
    "<doc>/reference/lambda.html"
    (form ((lib "racket/private/base.rkt") λ))
    "(form._((lib._racket/private/base..rkt)._~ce~bb))")
  #(syncheck:add-jump-to-definition/phase-level+space
    18
    20
    new-lambda
```

```
  "<collects>/racket/private/kw.rkt"
  ()
  0)
#(syncheck:add-mouse-over-status 18 20 "imported from racket")
#(syncheck:add-arrow/name-dup/pxpy
  10
  16
  0.5
  0.5
  18
  20
  0.5
  0.5
  #t
  0
  module-lang
  #<procedure:name-dup?>)
#(syncheck:add-arrow/name-dup/pxpy
  22
  23
  0.5
  0.5
  25
  26
  0.5
  0.5
  #t
  0
  #f
  #<procedure:name-dup?>)
#(syncheck:add-text-type 1 7 document-identifier)
#(syncheck:add-docs-menu
  1
  7
  module
  "View documentation for ‘‘module’’ from racket/base, racket"
  "<doc>/reference/module.html"
  (form ('#%kernel module))
  "(form._((quote._~23~25kernel)._module))")
#(syncheck:add-mouse-over-status 10 16 "1 bound occurrence")
#(syncheck:add-mouse-over-status 22 23 "1 bound occurrence"))
```

Changed in version 1.2 of package `drracket-tool-text-lib`: Added the `#:fully-expanded` and `#:namespace` arguments.

```
                                      (->* (syntax?)
(make-traversal namespace path) →          ((-> any/c void?))
                                           void?)
                                      (-> void?)
  namespace : namespace?
  path : (or/c #f path-string?)
```

This function creates some local state about a traversal of syntax objects and returns two functions. The first one should be called with each of the (fully expanded) syntax objects that make up a program (there will be only one if the program is a module) and then the second one should be called to indicate there are no more.

The optional argument to the first function is ignored. It is left there for historical reasons. In the past it was called for each sequence of binding identifiers encountered in `define-values`, `define-syntaxes`, and `define-values-for-syntax`.

During the dynamic extent of the call to the two result functions, the value of the `current-annotations` parameter is consulted and various methods are invoked in the corresponding object (if any), to indicate what has been found in the syntax object. These methods will only be called if the syntax objects have source locations.

The *path* argument indicates a directory whose traversal should operate on. When *path* is `#f`, it defaults to `(current-directory)`. Otherwise, the path is simplified via `simple-form-path` before it's used.

```
(current-annotations)
 → (or/c #f (is-a?/c syncheck-annotations<%>))
(current-annotations ca) → void?
  ca : (or/c #f (is-a?/c syncheck-annotations<%>))
```

The methods of the value of this parameter are invoked by the functions returned from `make-traversal`.

```
(current-max-to-send-at-once)
 → (or/c +inf.0 (and/c exact-integer? (>=/c 2)))
(current-max-to-send-at-once m) → void?
  m : (or/c +inf.0 (and/c exact-integer? (>=/c 2)))
```

No longer used.

```
syncheck-annotations<%> : interface?
```

Classes implementing this interface are acceptors of information about a traversal of syntax objects. See `make-traversal`.

Do not implement this interface directly, as it is liable to change without warning. Instead, use the `annotations-mixin` and override the methods you're interested in. The

annotations-mixin will keep in sync with this interface, providing methods that ignore their arguments.

```
(send a-syncheck-annotations syncheck:find-source-
object stx)
 → (or/c #f (not/c #f))
  stx : syntax?
```

This should return #f if the source of this syntax object is uninteresting for annotations (if, for example, the only interesting annotations are those in the original file and this is a syntax object introduced by a macro and thus has a source location from some other file).

Otherwise, it should return some (non-#f) value that will then be passed to one of the other methods below as a *source-obj* argument. Also, the result of this method is used as a key in an equal?-based hash, so two source objects should refer to the same original source if and only if they are equal?.

```
(send a-syncheck-annotations syncheck:add-text-type
 source-obj
 start
 end
 text-type)
 → void?
 source-obj : (not/c #f)
 start : exact-nonnegative-integer?
 end : exact-nonnegative-integer?
 text-type : (or/c 'matching-identifiers
                   'unused-identifier
                   'document-identifier)
```

Called to indicate that the color associated with the text type *text-type* should be drawn on the background of the given range in the editor, when the mouse moves over it.

This method is usually called by Check Syntax to add background colors to an identifier based on its lexical information. The types 'matching-identifiers, 'unused-identifier and 'document-identifier correspond to the color 'drracket:syncheck:matching-identifiers, 'drracket:syncheck:unused-identifier and 'drracket:syncheck:document-identifier in color scheme specifications, respectively. See §1.10 “Color Schemes”.

See syncheck:add-require-open-menu for information about the *start* and *end* arguments.

Added in version 1.8 of package drracket-tool-text-lib.

```
(send a-syncheck-annotations syncheck:add-background-color
 source-obj
 start
 end
 color)
→ void?
 source-obj : (not/c #f)
 start : exact-nonnegative-integer?
 end : exact-nonnegative-integer?
 color : string?
```

Called to indicate that the color *color* should be drawn on the background of the given range in the editor, when the mouse moves over it.

This method is not directly called by Check Syntax anymore. Instead see syncheck:add-text-type.

```
(send a-syncheck-annotations syncheck:add-require-open-menu
 source-obj
 start
 end
 file)
→ void?
 source-obj : (not/c #f)
 start : exact-nonnegative-integer?
 end : exact-nonnegative-integer?
 file : path-string?
```

Called to indicate that there is a `require` at the location from *start* to *end*, and that it corresponds to *file*.

The *start* and *end* coordinates typically come from a syntax object in the file that was processed (although they can be completely synthesized by macros in some situations). The *start* coordinate is one less than that syntax object's syntax-position field, and the *end* is the *start* plus that syntax-object's syntax-span field. Thus, it is always the case that `(<= start end)` is true. In some situations, it may be that *start* can equal *end*.

Check Syntax adds a popup menu.

```
(send a-syncheck-annotations syncheck:add-docs-menu
 source-obj
 start
 end
 id
 label
 definition-tag
 path
 tag)
→ void?
 source-obj : (not/c #f)
 start : exact-nonnegative-integer?
 end : exact-nonnegative-integer?
 id : symbol?
 label : any/c
 definition-tag : definition-tag?
 path : any/c
 tag : any/c
```

Called to indicate that there is something that has documentation between the range *start* and *end*. The documented identifier's name is given by *id* and the docs are found in the html file *path* at the html tag *tag*. The *definition-tag* argument matches the documented definition. The *label* argument describes the binding for use in the menu item (although it may be longer than 200 characters).

See syncheck:add-require-open-menu for information about the coordinates *start* and *end*.

```
(send a-syncheck-annotations syncheck:add-id-set
 all-ids
 new-name-interferes?)
→ void?
 all-ids : (listof (list/c (not/c #f)
                           exact-nonnegative-integer?
                           exact-nonnegative-integer?))
 new-name-interferes? : (-> symbol boolean?)
```

This method is no longer called by Check Syntax. It is here for backwards compatibility only. The information it provided must now be synthesized from the information supplied to syncheck:add-arrow/name-dup/pxpy.

```
(send a-syncheck-annotations syncheck:add-arrow
 start-source-obj
 start-left
 start-right
 end-source-obj
 end-left
 end-right
 actual?
 phase-level)
→ void?
 start-source-obj : (not/c #f)
 start-left : exact-nonnegative-integer?
 start-right : exact-nonnegative-integer?
 end-source-obj : (not/c #f)
 end-left : exact-nonnegative-integer?
 end-right : exact-nonnegative-integer?
 actual? : boolean?
 phase-level : (or/c exact-nonnegative-integer? #f)
```

This function is not called directly anymore by Check Syntax. Instead syncheck:add-arrow/name-dup/pxpy is.

This method is invoked by the default implementation of *syncheck:add-arrow/name-dup* in annotations-mixin.

See syncheck:add-require-open-menu for information about the coordinates; *start-left* and *start-right* are a pair like syncheck:add-require-open-menu's *start* and *end*, as are *end-left* and *end-right*.

```
(send a-syncheck-annotations syncheck:add-arrow/name-dup
 start-source-obj
 start-left
 start-right
 end-source-obj
 end-left
 end-right
 actual?
 phase-level
 require-arrow?
 name-dup?)
→ void?
 start-source-obj : (not/c #f)
 start-left : exact-nonnegative-integer?
 start-right : exact-nonnegative-integer?
 end-source-obj : (not/c #f)
 end-left : exact-nonnegative-integer?
 end-right : exact-nonnegative-integer?
```

```
 actual? : boolean?
 phase-level : (or/c exact-nonnegative-integer? #f)
 require-arrow? : boolean?
 name-dup? : (-> string? boolean?)
```

This function is not called directly anymore by Check Syntax. Instead syncheck:add-arrow/name-dup/pxpy is.

The default implementation of syncheck:add-arrow/name-dup/pxpy discards the *start-px start-py end-px end-py* arguments and calls this method.

See syncheck:add-require-open-menu for information about the coordinates; *start-left* and *start-right* are a pair like syncheck:add-require-open-menu's *start* and *end*, as are *end-left* and *end-right*.

```
(send a-syncheck-annotations syncheck:add-arrow/name-dup/pxpy
 start-source-obj
 start-left
 start-right
 start-px
 start-py
 end-source-obj
 end-left
 end-right
 end-px
 end-py
 actual?
 phase-level
 require-arrow
 name-dup?)
→ void?
 start-source-obj : (not/c #f)
 start-left : exact-nonnegative-integer?
 start-right : exact-nonnegative-integer?
 start-px : (real-in 0 1)
 start-py : (real-in 0 1)
 end-source-obj : (not/c #f)
 end-left : exact-nonnegative-integer?
 end-right : exact-nonnegative-integer?
 end-px : (real-in 0 1)
 end-py : (real-in 0 1)
 actual? : boolean?
 phase-level : (or/c exact-nonnegative-integer? #f)
 require-arrow : (or/c boolean? 'module-lang)
 name-dup? : (-> string? boolean?)
```

Called to indicate that there should be an arrow between the locations described by the first ten arguments. The *start-px* and *start-py* indicate how far along the diagonal between the upper-left coordinate of the editor position *start-left* and the bottom-right of the editor position *start-right* to draw the foot of the arrow. The *end-px* and *end-py* indicate the same things for the head of the arrow.

The *phase-level* argument indicates the phase of the binding and the *actual?* argument indicates if the binding is a real one, or a predicted one from a syntax template (predicted bindings are drawn with question marks in Check Syntax).

The *require-arrow* argument indicates if this arrow points from an imported identifier to its corresponding `require`. Any true value means that it points to an import via `require`; `#t` means it was a normal `require` and `'module-lang` means it comes from the implicit require that a module language provides.

The *name-dup?* predicate returns `#t` in case that this variable (either the start or end), when replaced with the given string, would shadow some other binding (or otherwise interfere with the binding structure of the program at the time the program was expanded).

See `syncheck:add-require-open-menu` for information about the coordinates; *start-left* and *start-right* are a pair like `syncheck:add-require-open-menu`'s *start* and *end*, as are *end-left* and *end-right*.

Changed in version 1.1 of package `drracket-tool-text-lib`: Changed *require-arrow* to sometimes be `'module-lang`.

```
(send a-syncheck-annotations syncheck:add-tail-arrow
 from-source-obj
 from-pos
 to-source-obj
 to-pos)
→ void?
 from-source-obj : (not/c #f)
 from-pos : exact-nonnegative-integer?
 to-source-obj : (not/c #f)
 to-pos : exact-nonnegative-integer?
```

Called to indicate that there are two expressions, beginning at *from-pos* and *to-pos* that are in tail position with respect to each other.

```
(send a-syncheck-annotations syncheck:add-mouse-over-status
 source-obj
 start
 end
 str)
→ void?
```

```
 source-obj : (not/c #f)
 start : exact-nonnegative-integer?
 end : exact-nonnegative-integer?
 str : string?
```

Called to indicate that the message in *str* should be shown when the mouse passes over a position in the given range between *start* and *end*.

See syncheck:add-require-open-menu for information about the coordinates *start* and *end*.

```
(send a-syncheck-annotations syncheck:add-prefixed-require-reference
 req-src
 req-pos-left
 req-pos-right
 prefix
 prefix-src
 prefix-left
 prefix-right)
→ void?
 req-src : (not/c #f)
 req-pos-left : exact-nonnegative-integer?
 req-pos-right : exact-nonnegative-integer?
 prefix : symbol?
 prefix-src : any/c
 prefix-left : (or/c #f exact-nonnegative-integer?)
 prefix-right : (or/c #f exact-nonnegative-integer?)
```

This method is called for each `require` in the program that has a *prefix* or *prefix-all-except* around it in fully expanded form (i.e., it seems to come from a `prefix-in` or a similar form).

The method is passed the location of the `require` in the original program, as well as the prefix (as a symbol) and the source locations of the prefix (if they are available).

See syncheck:add-require-open-menu for information about the coordinates; *req-pos-left* and *req-pos-right* are a pair like syncheck:add-require-open-menu's *start* and *end*, as are *prefix-left* and *prefix-right*.

```
(send a-syncheck-annotations syncheck:add-unused-require
 req-src
 start
 end)
→ void?
 req-src : (not/c #f)
 start : exact-nonnegative-integer?
 end : exact-nonnegative-integer?
```

This method is called for each `require` that Check Syntax determines to be unused. The method is passed the location of the name of the required module in the original program.

See `syncheck:add-require-open-menu` for information about the coordinates *start* and *end*.

```
(send a-syncheck-annotations syncheck:add-jump-to-definition
 source-obj
 start
 end
 id
 filename
 submods)
→ void?
 source-obj : (not/c #f)
 start : exact-nonnegative-integer?
 end : exact-nonnegative-integer?
 id : any/c
 filename : path-string?
 submods : (listof symbol?)
```

This function is not called directly anymore by Check Syntax. Instead `syncheck:add-jump-to-definition/phase-level+space` is.

The default implementation of `syncheck:add-jump-to-definition/phase-level+space` discards the *phase-level* argument and calls this method.

```
(send a-syncheck-annotations syncheck:add-jump-to-definition/phase-level+space
 source-obj
 start
 end
 id
 filename
 submods
 phase-level+space)
→ void?
 source-obj : (not/c #f)
 start : exact-nonnegative-integer?
 end : exact-nonnegative-integer?
 id : any/c
 filename : path-string?
 submods : (listof symbol?)
 phase-level+space : phase+space-shift?
```

Called to indicate that there is some identifier at the given location (named *id*) that is defined in the *submods* of the file *filename* (where an empty list in

submods means that the identifier is defined at the top-level module), at the phase level phase-level.

See syncheck:add-require-open-menu for information about the coordinates *start* and *end*.

Added in version 1.2 of package drracket-tool-text-lib.

```
(send a-syncheck-annotations syncheck:add-definition-target
 source-obj
 start
 finish
 id
 mods)
→ void?
 source-obj : (not/c #f)
 start : exact-nonnegative-integer?
 finish : exact-nonnegative-integer?
 id : symbol?
 mods : (listof symbol?)
```

This function is not called directly anymore by Check Syntax. Instead syncheck:add-definition-target/phase-level+space is.

The default implementation of syncheck:add-definition-target/phase-level+space discards the *phase-level* argument and calls this method.

```
(send a-syncheck-annotations syncheck:add-definition-target/phase-level+space
 source-obj
 start
 finish
 id
 mods
 phase-level)
→ void?
 source-obj : (not/c #f)
 start : exact-nonnegative-integer?
 finish : exact-nonnegative-integer?
 id : symbol?
 mods : (listof symbol?)
 phase-level : phase+space-shift?
```

Called to indicate a top-level definition at the location spanned by *start* and *finish*. The *id* argument is the name of the defined variable and the *mods* are the submodules enclosing the definition, which will be empty if the definition is in the top-level module. The *phase-level* argument indicates the phase level.

See syncheck:add-require-open-menu for information about the coordinates *start* and end.

Added in version 1.2 of package drracket-tool-text-lib.

```
(send a-syncheck-annotations syncheck:unused-binder
 source-obj
 left
 right)
→ void?
 source-obj : (not/c #f)
 left : exact-nonnegative-integer?
 right : exact-nonnegative-integer?
```

Called to indicate that there is a binder with no references in *source-obj* at the location spanned by *left* and *right*.

Added in version 1.4 of package drracket-tool-text-lib.

```
(send a-syncheck-annotations syncheck:color-range source-obj
                                                   start
                                                   end
                                                   style-name)
→ void?
 source-obj : (not/c #f)
 start : exact-nonnegative-integer?
 end : exact-nonnegative-integer?
 style-name : any/c
```

Called to indicate that the given location should be colored according to the style *style-name*.

See syncheck:add-require-open-menu for information about the coordinates *start* and *end*.

```
(send a-syncheck-annotations syncheck:add-rename-menu
 id
 all-ids
 new-name-interferes?)
→ void?
 id : symbol?
 all-ids : (listof (list/c (not/c #f)
                           exact-nonnegative-integer?
                           exact-nonnegative-integer?))
 new-name-interferes? : (-> symbol boolean?)
```

This method is listed only for backwards compatibility. It is not called by Check Syntax anymore.

annotations-mixin : (class? . -> . class?)
  result implements: syncheck-annotations<%>

Supplies all of the methods in syncheck-annotations<%> with default behavior. Be sure to use this mixin to future-proof your code and then override the methods you're interested in.

By default:

- The syncheck:find-source-object method ignores its arguments and returns #f;
- the syncheck:add-arrow/name-dup method drops the *require-arrow?* and *name-dup?* arguments and calls syncheck:add-arrow;
- the syncheck:add-arrow/name-dup/pxpy method drops the *from-px*, *from-py*, *to-px*, and *to-py* arguments and calls syncheck:add-arrow/name-dup; and
- all of the other methods ignore their arguments and return (void).

Here is an example showing how use this library to extract all of the arrows that Check Syntax would draw from various expressions. One subtle point: arrows are only included when the corresponding identifiers are syntax-original?; the code below manages this by copying the properties from an identifier that is syntax-original? in the call to datum->syntax.

```
> (define arrows-collector%
    (class (annotations-mixin object%)
      (super-new)
      (define/override (syncheck:find-source-object stx)
        stx)
      (define/override (syncheck:add-arrow/name-dup/pxpy
                        start-source-obj start-left start-
right start-px start-py
                        end-source-obj end-left end-right end-
px end-py
                        actual? phase-level require-arrow? name-
dup?)
        (set! arrows
              (cons (list start-source-obj end-source-obj)
                    arrows)))
      (define arrows '())
      (define/public (get-collected-arrows) arrows)))
> (define (arrows form)
    (define base-namespace (make-base-namespace))
    (define-values (add-syntax done)
```

```
      (make-traversal base-namespace #f))
    (define collector (new arrows-collector%))
    (parameterize ([current-annotations collector]
                   [current-namespace base-namespace])
      (add-syntax (expand form))
      (done))
    (send collector get-collected-arrows))
> (define (make-id name pos orig?)
    (datum->syntax
     #f
     name
     (list #f #f #f pos (string-length (symbol->string name)))
     (and orig? #'is-orig)))
> (arrows `(λ (,(make-id 'x 1 #t)) ,(make-id 'x 2 #t)))
'((#<syntax x> #<syntax x>))
> (arrows `(λ (x) x))
'()
> (arrows `(λ (,(make-id 'x 1 #f)) ,(make-id 'x 2 #t)))
'()
> (arrows `(λ (,(make-id 'x 1 #t)) x))
'()
```

syncheck:find-source-object

Bound to an identifier created with `define-local-member-name` that is used as the `syncheck:find-source-object` method of `syncheck-annotations<%>`.

syncheck:add-text-type

Bound to an identifier created with `define-local-member-name` that is used as the `syncheck:add-text-type` method of `syncheck-annotations<%>`.

syncheck:add-background-color

Bound to an identifier created with `define-local-member-name` that is used as the `syncheck:add-background-color` method of `syncheck-annotations<%>`.

syncheck:add-require-open-menu

Bound to an identifier created with `define-local-member-name` that is used as the `syncheck:add-require-open-menu` method of `syncheck-annotations<%>`.

`syncheck:add-docs-menu`

Bound to an identifier created with `define-local-member-name` that is used as the `syncheck:add-docs-menu` method of `syncheck-annotations<%>`.

`syncheck:add-rename-menu`

Bound to an identifier created with `define-local-member-name` that is used as the `syncheck:add-rename-menu` method of `syncheck-annotations<%>`.

`syncheck:add-arrow`

Bound to an identifier created with `define-local-member-name` that is used as the `syncheck:add-arrow` method of `syncheck-annotations<%>`.

`syncheck:add-arrow/name-dup`

Bound to an identifier created with `define-local-member-name` that is used as the `syncheck:add-arrow/name-dup` method of `syncheck-annotations<%>`.

`syncheck:add-arrow/name-dup/pxpy`

Bound to an identifier created with `define-local-member-name` that is used as the `syncheck:add-arrow/name-dup/pxpy` method of `syncheck-annotations<%>`.

`syncheck:add-tail-arrow`

Bound to an identifier created with `define-local-member-name` that is used as the `syncheck:add-tail-arrow` method of `syncheck-annotations<%>`.

`syncheck:add-mouse-over-status`

Bound to an identifier created with `define-local-member-name` that is used as the `syncheck:add-mouse-over-status` method of `syncheck-annotations<%>`.

`syncheck:add-jump-to-definition`

Bound to an identifier created with `define-local-member-name` that is used as the `syncheck:add-jump-to-definition` method of `syncheck-annotations<%>`.

`syncheck:add-jump-to-definition/phase-level+space`

Bound to an identifier created with `define-local-member-name` that is used as the `syncheck:add-jump-to-definition/phase-level+space` method of `syncheck-annotations<%>`.

`syncheck:add-definition-target`

Bound to an identifier created with `define-local-member-name` that is used as the `syncheck:add-definition-target` method of `syncheck-annotations<%>`.

`syncheck:add-definition-target/phase-level+space`

Bound to an identifier created with `define-local-member-name` that is used as the `syncheck:add-definition-target/phase-level+space` method of `syncheck-annotations<%>`.

`syncheck:unused-binder`

Bound to an identifier created with `define-local-member-name` that is used as the `syncheck:unused-binder` method of `syncheck-annotations<%>`.

`syncheck:add-id-set`

Bound to an identifier created with `define-local-member-name` that is used as the `syncheck:add-id-set` method of `syncheck-annotations<%>`.

`syncheck:color-range`

Bound to an identifier created with `define-local-member-name` that is used as the `syncheck:color-range` method of `syncheck-annotations<%>`.

`syncheck:add-prefixed-require-reference`

Bound to an identifier created with `define-local-member-name` that is used as the `syncheck:add-prefixed-require-reference` method of `syncheck-annotations<%>`.

```
syncheck:add-unused-require
```

Bound to an identifier created with `define-local-member-name` that is used as the `syncheck:add-unused-require` method of `syncheck-annotations<%>`.

2 Module Browser

```
(require drracket/module-browser)
```
package: drracket-tool-lib

```
(module-browser path) → void?
  path : path-string?
```

Opens a window containing the module browser for *path*.

3 Module Path Selection

DrRacket provides two APIs for prompting the user to select a module path. One that uses the `racket/gui` library with a dialog box and one, lower-level, for use with another UI that provides just the information needed for completions.

3.1 GUI Module Path Selection

```
(require drracket/get-module-path)
```
package: drracket-tool-lib

```
(get-module-path-from-user
 [#:init init
  #:pref pref
  #:dir? dir?]
  #:current-directory current-directory)
→ (if dir?
      (or/c (listof path?) #f)
      (or/c path? #f))
 init : string? = ""
 pref : (or/c symbol? #f) = #f
 dir? : boolean? = #f
 current-directory : (or/c path-string? #f)
```

Opens a dialog box that facilitates choosing a path in the file system accessible via a module.

The user types a partial require path into the dialog and is shown completions of the require path and which paths they correspond to. (The initial content of the field where the user types is *init*.) Selecting one of the completions causes this function to return with the path of the selected one. If the *dir?* argument is `#t`, then the require path might not be complete, in which case the result is a list of directory paths corresponding to the directories where the partial require paths points. If the result is `#f`, then the user canceled the dialog.

The dialog also has an optional field where the path to some different racket binary than the one currently running. If that is filled in, then the dialog shows completions corresponding to how `require` would behave in that other racket binary. When that text field is edited, the *pref* is used with `preferences:set` and `preferences:get` to record its value so it persists across calls to `get-module-path-from-user`.

3.2 Module Path Selection Completion Computation

```
(require drracket/find-module-path-completions)
```
package: drracket-tool-text-lib

```
(find-module-path-completions dir)
 → (-> string? (listof (list/c string? path?)))
  dir : path-string?
```

This is the completion computing function for `get-module-path-from-user`.

The *dir* argument specifies a directory for relative require paths.

The result is a function that closes over some local state that is used to cache information to speed up repeated queries. (This cache should not be used across interactions with the user as it captures details about the current file system's directory and file layout.)

The result function's argument is the string the user has typed and the the result function's result is a new set of completions. Each element of the list corresponds to a completion. The `string?` portion of each element is the complete require and the `path?` portion is the path it matches in the filesystem. The `get-module-path-from-user` function shows the strings to the user and uses the paths to decide how to handle "return" keystrokes (and clicking on the "OK" button). If the path is a directory, then a "return" keystroke with descend into that directory (replacing the place where the user typed with the string portion of that element). If the path is not a directory, then return closes the dialog and returns the path.

Use `path->relative-string/library` to turn the paths into strings to show the user as potential completions.

```
(find-module-path-completions/explicit-cache
 str
 dir
 #:pkg-dirs-cache pkg-dirs-cache
 [#:alternate-racket alternate-racket])
→ (listof (list/c string? path?))
 str : string?
 dir : path-string?
 pkg-dirs-cache : (box/c (or/c #f pkg-dirs-info/c))
 alternate-racket : (or/c #f
                          path-string?
                          (list/c
                           current-library-collection-links-info/c
                           current-library-collection-paths-info/c
                           pkg-dirs-info/c))
                  = #f
```

This is a version of `find-module-path-completions` that explicates the *pkg-dirs-cache* argument and supports using a different racket binary (as discussed in `get-module-path-from-user`).

The *pkg-dirs-cache* argument should initially be `(box #f)`; it is filled in with the cached

information and then the filled in box can be used on subsequent calls.

Use `alternate-racket-clcl/clcp` to get the values for the *alternate-racket* argument in the case that an alternate racket is used. Pass `#f` for the current racket.

```
(alternate-racket-clcl/clcp alternate-racket
                            pkg-dirs-cache)
 → current-library-collection-links-info/c
   current-library-collection-paths-info/c
   pkg-dirs-info/c
  alternate-racket : path-string?
  pkg-dirs-cache : (box/c (or/c #f pkg-dirs-info/c))
```

Computes the information needed for completions by calling out to the external racket binary *alternate-racket*. Use the same *pkg-dirs-cache* argument as with `find-module-path-completions/explicit-cache`.

```
current-library-collection-links-info/c : contract?
```

A contract specifying what information used by this library relevant to the current library links.

```
current-library-collection-paths-info/c : contract?
```

A contract specifying what information used by this library relevant to the current library collections.

```
pkg-dirs-info/c : contract?
```

A contract specifying what information used by this library relevant to the pkg directories.