Define2: Enhancing define

Laurent Orseau

RacketFest — March 2023

Arity errors

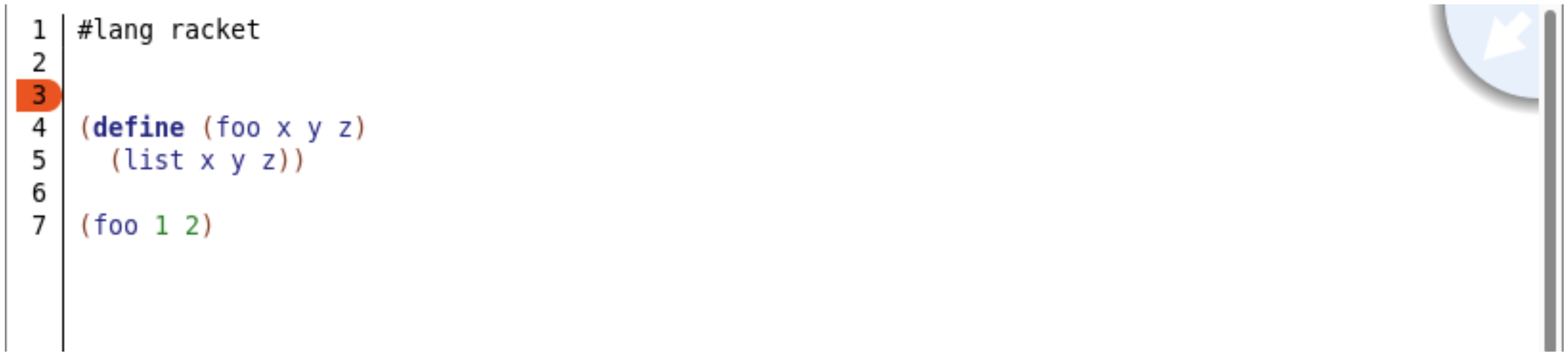
- When is the error raised?

```
(define (foo x y)
  (println (list x y)))

(for ([i 6])
  (sleep 10)
  (if (< i 5)
    (foo 1 2)
    (foo 3)))
```

Compile-time checks

Catch arity errors at compile-time

- Racket's define:

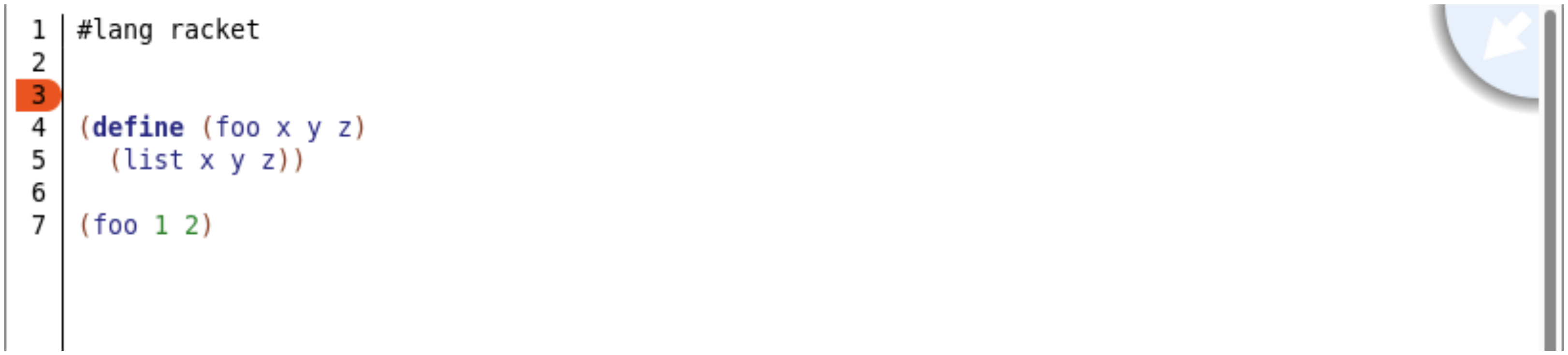


```
#lang racket


(define (foo x y z)
  (list x y z))

(foo 1 2)
```

Compile-time checks

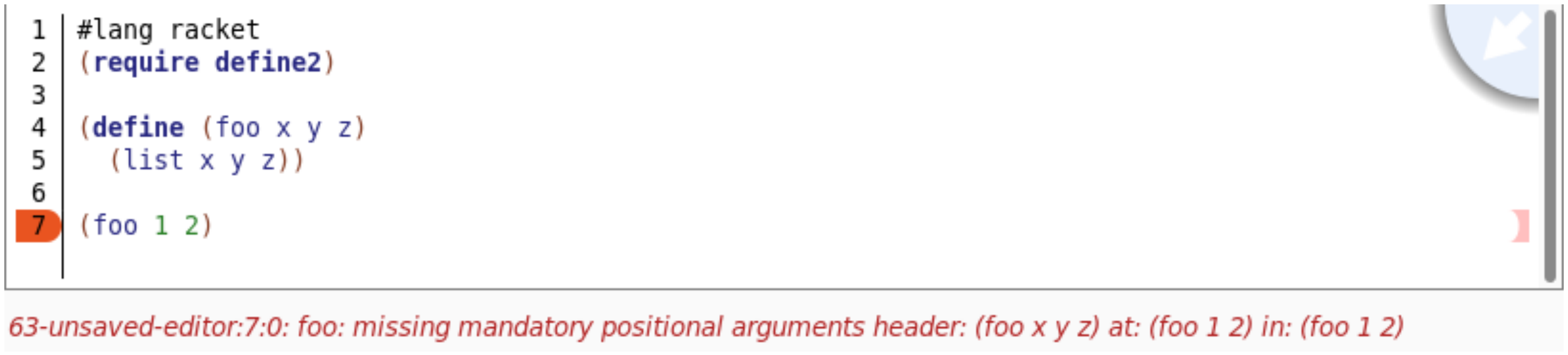
Catch arity errors at compile-time

- Racket's define:



```
#lang racket


(define (foo x y z)
  (list x y z))

(foo 1 2)
```

- define2:



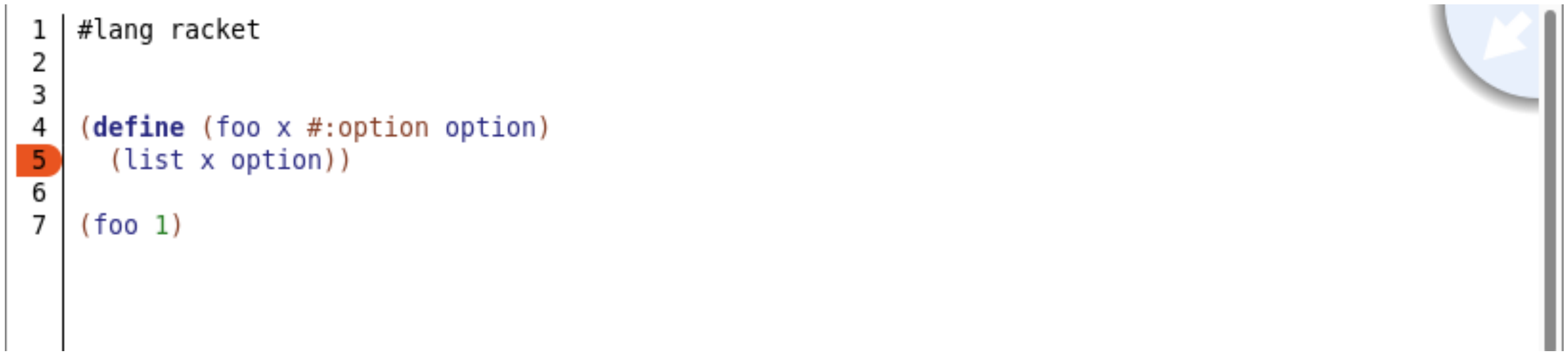
```
#lang racket
(require define2)

(define (foo x y z)
  (list x y z))

(foo 1 2)
```

```
63-unsaved-editor:7:0: foo: missing mandatory positional arguments header: (foo x y z) at: (foo 1 2) in: (foo 1 2)
```

Compile-time checks

Catch keyword errors at compile-time

- Racket's define:

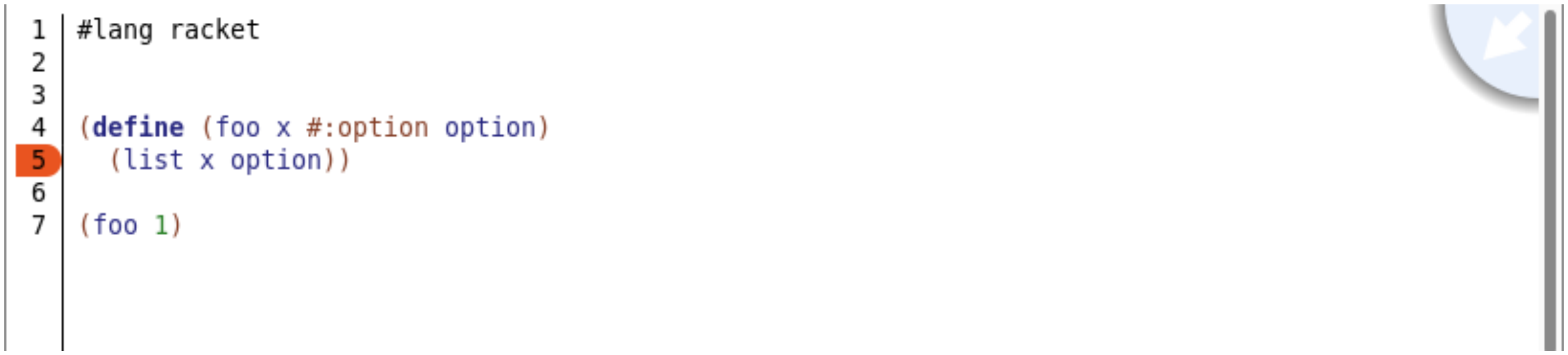


```
#lang racket


(define (foo x #:option option)
  (list x option))

(foo 1)
```

Compile-time checks

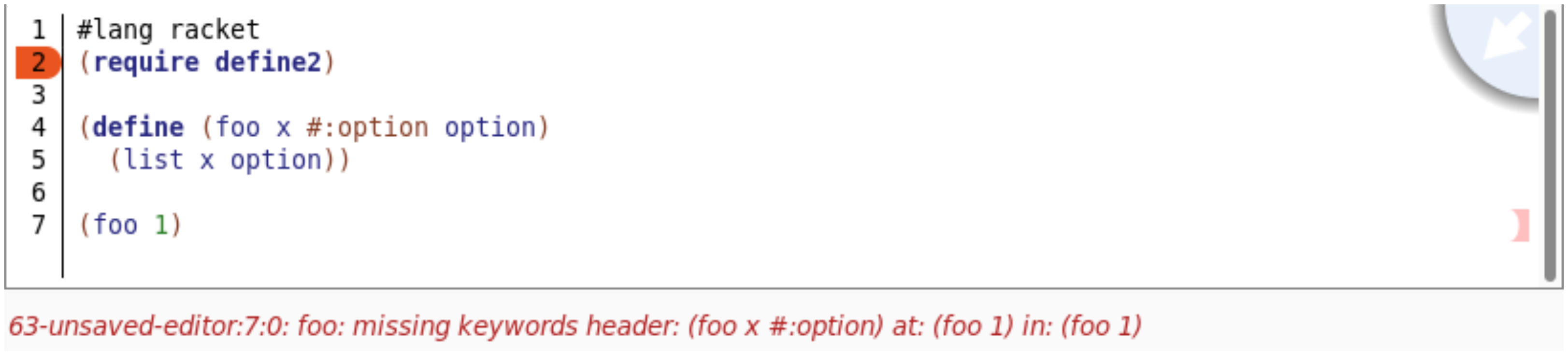
Catch keyword errors at compile-time

- Racket's define:



```
#lang racket


(define (foo x #:option option)
  (list x option))

(foo 1)
```

- define2:



```
#lang racket
(require define2)

(define (foo x #:option option)
  (list x option))

(foo 1)
```

63-unsaved-editor:7:0: foo: missing keywords header: (foo x #:option) at: (foo 1) in: (foo 1)

Curried functions

- Compile-time check for curried functions at first level only

```
(define ((foo x) y)
  (list x y))
```

Curried functions

- Compile-time check for curried functions at first level only

```
(define ((foo x) y)
  (list x y))

(foo) ; raises compile-time exn
```

Curried functions

- Compile-time check for curried functions at first level only

```
(define ((foo x) y)
  (list x y))

(foo) ; raises compile-time exn

((foo 2)) ; no compile-time exn
```

Simplifying keyword arguments: Mandatory arguments

- Keywords make intention very clear

```
(geometry 2 3 2 4 5 5)
```

vs

Simplifying keyword arguments: Mandatory arguments

- Keywords make intention very clear

```
(geometry 2 3 2 4 5 5)
```

vs

```
(geometry #:x-bottom-left 2 #:y-bottom-left 3
          #:x-top-right   2 #:y-top-right   4
          #:x-margin      5 #:y-margin      5)
```

Simplifying keyword arguments: Mandatory arguments

- Keywords make intention very clear

```
(geometry 2 3 2 4 5 5)
```

vs

```
(geometry #:x-bottom-left 2 #:y-bottom-left 3
          #:x-top-right   2 #:y-top-right   4
          #:x-margin      5 #:y-margin      5)
```

- But too much repetition with **`define`**

```
(define (geometry #:x-bottom-left x-bottom-left #:y-bottom-left y-bottom-left
                  #:x-top-right   x-top-right   #:y-top-right   y-top-right
                  #:x-margin      x-margin      #:y-margin      y-margin)
  ...)
```

Simplifying keyword arguments: Mandatory arguments

- With define2: `#:! id`

Simplifying keyword arguments: Mandatory arguments

- With define2: `#:! id`

```
(define (geometry #:! x-bottom-left #:! y-bottom-left
                  #:! x-top-right   #:! y-top-right
                  #:! x-margin      #:! y-margin)
  ...)
```

Simplifying keyword arguments: Mandatory arguments

- With define2: `#:! id`

```
(define (geometry #:! x-bottom-left #:! y-bottom-left
                  #:! x-top-right   #:! y-top-right
                  #:! x-margin      #:! y-margin)
  ...)
```

- Calls don’t change

```
(geometry #:x-bottom-left 2 #:y-bottom-left 3
          #:x-top-right   2 #:y-top-right   4
          #:x-margin      5 #:y-margin      5)
```

Simplifying keyword arguments: Optional arguments

- `#:? [id default-val]`

Simplifying keyword arguments: Optional arguments

- `#:? [id default-val]`

```
(define (rectangle #:! x #:! y #:! width #:? [height width])
  ...)

(rectangle #:x 0 #:y 0 #:width 10)
```

Simplifying keyword arguments: Optional arguments

- `#:? [id default-val]`

```
(define (rectangle #:! x #:! y #:! width #:? [height width])
  ...)

(rectangle #:x 0 #:y 0 #:width 10)
```

- old style still works: `#:kw-id [new-id val]`

Default argument issues

- Let's write a wrapper for `dict-ref`

```
(define d '(…))
(define (d-ref key [default ???])
  (dict-ref d key default))
```

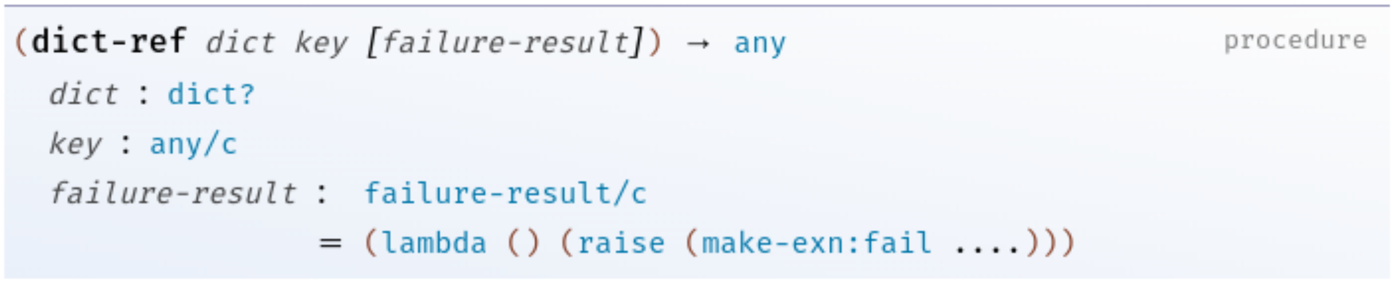
- What is dict-ref's default?

Default argument issues

- Let’s write a wrapper for **`dict-ref`**

```
(define d '(…))
(define (d-ref key [default ???])
  (dict-ref d key default))
```

- What is dict-ref’s default?



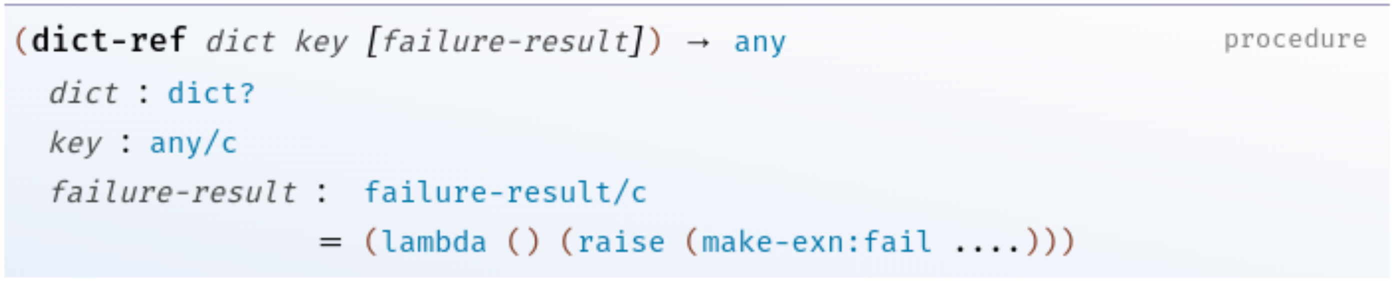
```
(dict-ref dict key [failure-result]) → any                    procedure
  dict : dict?
  key : any/c
  failure-result :  failure-result/c
                 = (lambda () (raise (make-exn:fail ....)))
```

Default argument issues

- Let's write a wrapper for `dict-ref`

```
(define d '(…))
(define (d-ref key [default ???])
  (dict-ref d key default))
```

- What is dict-ref's default?



```
(dict-ref dict key [failure-result]) → any                    procedure
  dict : dict?
  key : any/c
  failure-result :  failure-result/c
                 = (lambda () (raise (make-exn:fail ....)))
```

LIES! 😱

Wrapping dict-ref

- private/dict.rkt:

```
(define no-arg (gensym))
(define (assoc-ref d key [default no-arg])
  …)
```

Wrapping dict-ref

- private/dict.rkt:

```
(define no-arg (gensym))
(define (assoc-ref d key [default no-arg])
  (cond
    [(eq? default no-arg)
     (raise-mismatch-error
      'dict-ref
      (format "no value for key: ~e in: " key)
      d)]
    …))
```

Wrapping dict-ref

- private/dict.rkt:

```
(define no-arg (gensym))
(define (assoc-ref d key [default no-arg])
  (cond
    [(eq? default no-arg)
     (raise-mismatch-error
      'dict-ref
      (format "no value for key: ~e in: " key)
      d)]
    …))
```

`no-arg` not exported 😕

Wrapping dict-ref

- define2 generalizes the use of no-value
    - pass-through optional arguments

Wrapping dict-ref

- define2 generalizes the use of no-value
  - pass-through optional arguments
  - no-value is exported but usually not needed

Wrapping dict-ref

- define2 generalizes the use of no-value
  - pass-through optional arguments
  - no-value is exported but usually not needed

```
(require define2)
(define (dict-ref d key #:? [default (λ () (error …))])
  …)
…
(define d '(…))
(define (d-ref key #:? default) ; default = no-value
  (dict-ref d key #:default default))
```

Pass-through optional arguments

```
(define (rectangle #:! x #:! y #:! width #:? [height width])
  ...)
```

Pass-through optional arguments

```
(define (rectangle #:! x #:! y #:! width #:? [height width])
  ...)

(define (rectangle10 #:! x #:! y #:? [width 10] #:? height)
  (rectangle #:x x #:y y #:width width #:height height))
```

Pass-through optional arguments

```
(define (rectangle #:! x #:! y #:! width #:? [height width])
  ...)

(define (rectangle10 #:! x #:! y #:? [width 10] #:? height)
  (rectangle #:x x #:y y #:width width #:height height))

(rectangle10 #:x 1 #:y 2)
```

Pass-through implementation

- **`no-value`**: Like dict-ref's no-arg

```
(define (foo #:? [a 5] #:? b)
  …)
```

Pass-through implementation

- `no-value`: Like dict-ref's no-arg

```
(define (foo #:? [a 5] #:? b)
  …)

; equivalent to:

(define (foo #:a [a no-value] #:b [b no-value])
  (let* ([a (if (eq? a no-value) 5 a)])
    …))
```

Wrapping plot

- I want a function like plot, but …

Wrapping plot

- I want a function like plot, but …
  - with different default arguments

Wrapping plot

- I want a function like plot, but …
  - with different default arguments
  - with less/more arguments

Wrapping plot

- I want a function like plot, but …
  - with different default arguments
  - with less/more arguments
  - with pre/post processing

Wrapping plot

- I want a function like plot, but …
  - with different default arguments
  - with less/more arguments
  - with pre/post processing
- parameters ok, but not a `plot` function

Wrapping plot

- I want a function like plot, but …
  - with different default arguments
  - with less/more arguments
  - with pre/post processing
- parameters ok, but not a `plot` function
- make-keyword-procedure + keyword-apply
  - too permissive
  - too low level

Wrapping plot

- Manual wrapping with Racket’s define

```
(define (my-plot1 renderer-tree
                  #:x-min [x-min #f] #:x-max [x-max #f]
                  #:y-min [y-min #f] #:y-max [y-max #f]
                  #:width [width (plot-width)]
                  #:height [height (plot-height)]
                  #:title [title (plot-title)]
                  #:x-label [x-label (plot-x-label)]
                  #:y-label [y-label (plot-y-label)]
                  #:aspect-ratio [aspect-ratio (plot-aspect-ratio)]
                  #:legend-anchor [legend-anchor (plot-legend-anchor)]
                  #:out-file [out-file #f]
                  #:out-kind [out-kind 'auto])
  (plot renderer-tree
        #:x-min x-min #:x-max x-max
        #:y-min y-min #:y-max y-max
        #:width width
        #:height height
        #:title title
        #:x-label x-label
        #:y-label y-label
        #:aspect-ratio aspect-ratio
        #:legend-anchor legend-anchor
        #:out-file out-file
        #:out-kind out-kind))
```

😭

Wrapping plot

- Manual wrapping with define2

```
(define (plot1 renderer-tree
                  #:? [x-min #f] #:? [x-max #f]
                  #:? [y-min #f] #:? [y-max #f]
                  #:? [width (plot-width)]
                  #:? [height (plot-height)]
                  #:? [title (plot-title)]
                  #:? [x-label (plot-x-label)]
                  #:? [y-label (plot-y-label)]
                  #:? [aspect-ratio (plot-aspect-ratio)]
                  #:? [legend-anchor (plot-legend-anchor)]
                  #:? [out-file #f]
                  #:? [out-kind 'auto])
  (plot renderer-tree
        #:x-min x-min #:x-max x-max
        #:y-min y-min #:y-max y-max
        #:width width
        #:height height
        #:title title
        #:x-label x-label
        #:y-label y-label
        #:aspect-ratio aspect-ratio
        #:legend-anchor legend-anchor
        #:out-file out-file
        #:out-kind out-kind))
```

😢

Wrapping plot

- define2/define-wrapper

```
(define-wrapper (plot2
                 (plot renderer-tree
                       #:? [x-min #f] #:? [x-max #f]
                       #:? [y-min #f] #:? [y-max #f]
                       #:? [width   (plot-width)]
                       #:? [height  (plot-height)]
                       #:? [title   (plot-title)]
                       #:? [x-label (plot-x-label)]
                       #:? [y-label (plot-y-label)]
                       #:? [aspect-ratio  (plot-aspect-ratio)]
                       #:? [legend-anchor (plot-legend-anchor)]
                       #:? [out-file #f]
                       #:? [out-kind 'auto])))
```

😌

Wrapping plot

- Wrapping `plot2` even easier
  - Or: if plot was defined with define2

```
(define-wrapper
  (my-plot (plot2 renderer-tree
                    #:? [x-label "Zee x-axis"]
                    #:? [y-label "Zee y-axis"]
                    ;  Pass-through arguments
                    #:? x-min #:? x-max #:? width #:? height
                    #:? title #:? aspect-ratio #:? legend-anchor
                    #:? out-file #:? out-kind)))
```

Wrapping plot

- Wrapping `plot2` even easier
  - Or: if plot was defined with define2

```
(define-wrapper
  (my-plot (plot2 renderer-tree
                  #:? [x-label "Zee x-axis"]
                  #:? [y-label "Zee y-axis"]
                  ;  Pass-through arguments
                  #:? x-min #:? x-max #:? width #:? height
                  #:? title #:? aspect-ratio #:? legend-anchor
                  #:? out-file #:? out-kind)))

(my-plot (function sqr 0 1))
```

Wrapping plot

- Wrapping **`plot2`** even easier
  - Or: if plot was defined with define2

```
(define-wrapper
  (my-plot (plot2 renderer-tree
                  #:? [x-label "Zee x-axis"]
                  #:? [y-label "Zee y-axis"]
                  ;  Pass-through arguments
                  #:? x-min #:? x-max #:? width #:? height
                  #:? title #:? aspect-ratio #:? legend-anchor
                  #:? out-file #:? out-kind)))

(my-plot (function sqr 0 1))

(my-plot (function sqr 0 1) #:y-label "Why?!")
```

Wrapping plot

- Wrapping `plot2` even easier
  - Or: if plot was defined with define2

```
(define-wrapper
  (my-plot (plot2 renderer-tree
                  #:? [x-label "Zee x-axis"]
                  #:? [y-label "Zee y-axis"]
                  ;  Pass-through arguments
                  #:? x-min #:? x-max #:? width #:? height
                  #:? title #:? aspect-ratio #:? legend-anchor
                  #:? out-file #:? out-kind)))

(my-plot (function sqr 0 1))

(my-plot (function sqr 0 1) #:y-label "Why?!")

(my-plot (function sqr 0 1) #:y-label "Why?!" #:x-max 20)
```

Wrapping + pre/post processing

```
(define-wrapper (plot+time (my-plot2 renderer-tree
                                     #:? x-label #:? y-label
                                     #:? x-min #:? x-max #:? width #:? height
                                     #:? title #:? aspect-ratio #:? legend-anchor
                                     #:? out-file #:? out-kind))
  #:call-wrapped call-me
  (define before (current-milliseconds))
  (define the-plot (call-me))
  (define after (current-milliseconds))
  (values the-plot (- after before)))
```

Implementation of define2

- syntax-parse
- syntax classes
- Adapted from syntax/parse/lib/function-header.rkt
- helped from several people
  - Likely:
    Alexis King, Bogdan Popa, Sorawee Porncharoenwase, Jack Firth, Sam Tobin-Hochstadt, Matthew Flatt, Bogdan Popa, Jens Axel Søgaard …

Conclusion

- define2 almost fully backward compatible
  - Unless you already use `#:?` or `#:!`

Conclusion

- define2 almost fully backward compatible
  - Unless you already use `#:?` or `#:!`
- Compile time arity checks

Conclusion

- define2 almost fully backward compatible
  - Unless you already use `#:?` or `#:!`
- Compile time arity checks
- Simplifies keyword arguments
  - Lowers the cognitive barrier
  - Reduces the visual load

Conclusion

- define2 almost fully backward compatible
  - Unless you already use `#:?` or `#:!`
- Compile time arity checks
- Simplifies keyword arguments
  - Lowers the cognitive barrier
  - Reduces the visual load
- Pass-through arguments
  - No need to know default argument values of primitives

Conclusion

- define2 almost fully backward compatible
  - Unless you already use `#:?` or `#:!`
- Compile time arity checks
- Simplifies keyword arguments
  - Lowers the cognitive barrier
  - Reduces the visual load
- Pass-through arguments
  - No need to know default argument values of primitives
- Wrapping utilities for functions of many arguments