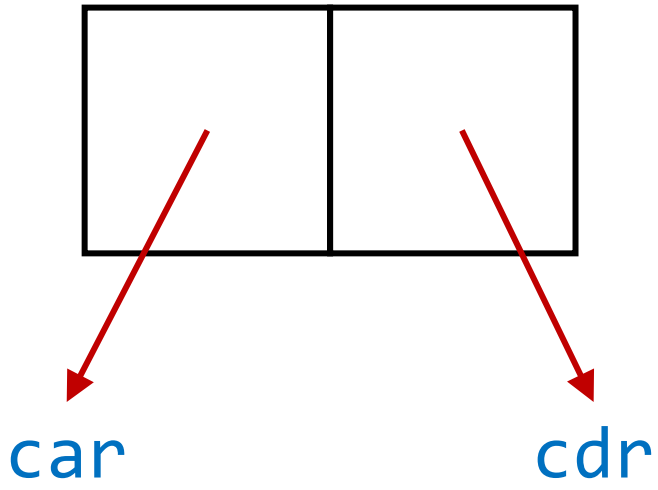
NOTE: You don't need to know box-pointer notation! It's simply a useful visualization.

Pairs (Box-Pointer Notation)

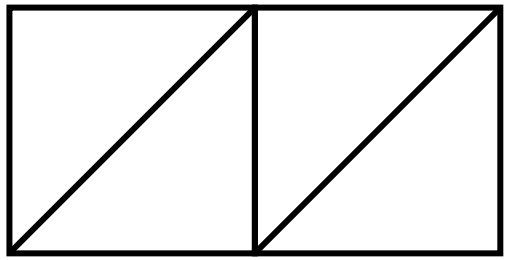
A pair is represented by two connected boxes. Each box contains a pointer; the left points to the `car` of the pair, and the right to the `cdr`.



A pair is *constructed* in Racket with `(cons a d)`, and the pair datatype is represented with a period.

Ex: `(cons 1 2)` yields `'(1 . 2)`

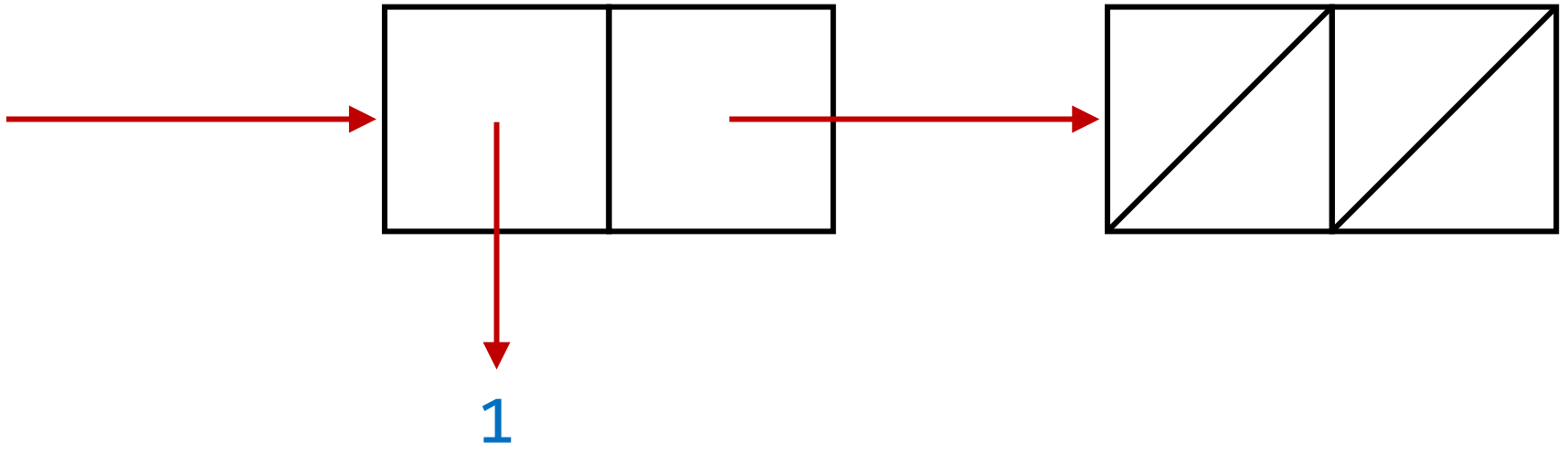
If a pointer in the pair is pointing to nothing (null), we can either show that with an arrow pointing to nothing, or in this case we'll draw a line through the pointer's box.



The above diagram represents the empty list, written in Racket as `'()`.

Lists in Racket are implemented as linked lists, so we build a list by chaining pointers together, with the empty list at the end.

Building Lists from Pairs

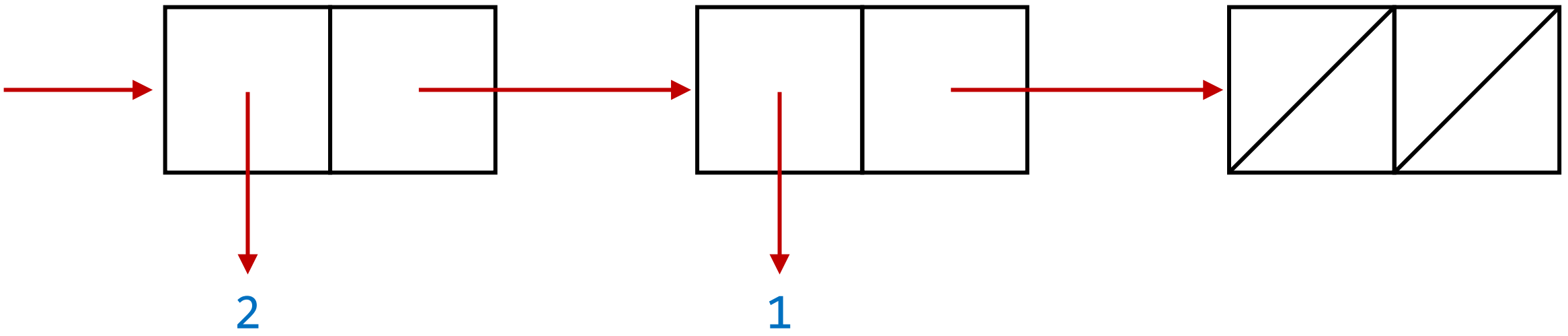


The above diagram represents the list `'(1)`, a.k.a. `(list 1)`, or more literally `(cons 1 '())`.
When we `cons` something with a list rather than with an item, the result is represented as a list in Racket.

Note that the first pair's right pointer is pointing to the entire second pair, not its left box. You can't point to one of a pair's boxes individually.

The left-most arrow indicates the pointer from the external context (whatever program this list is embedded in) to the start of the list.

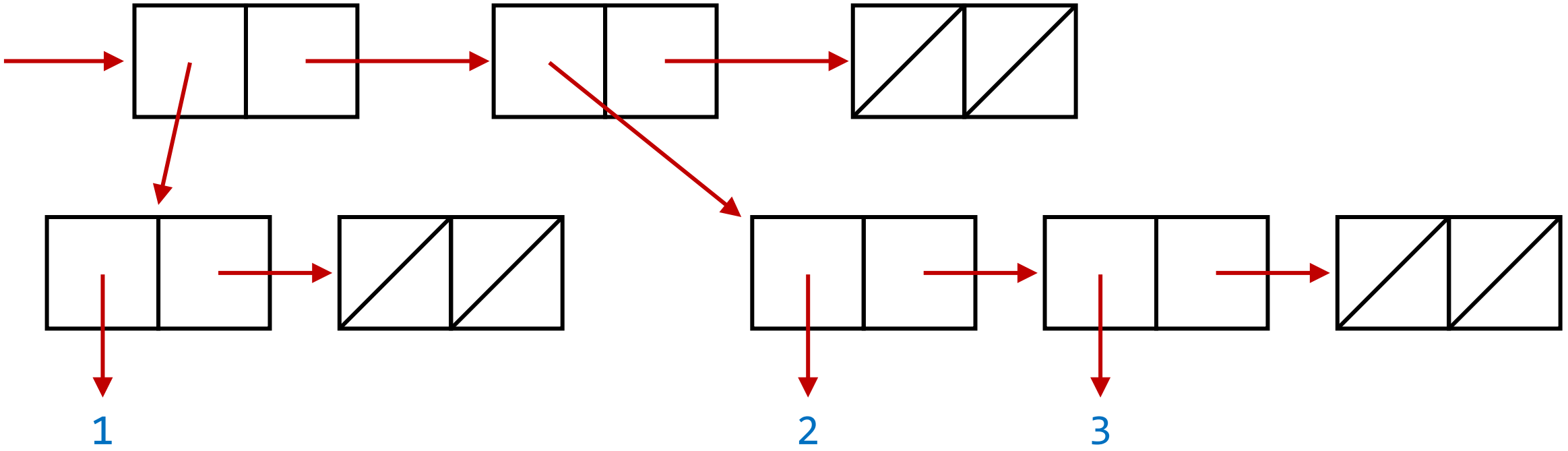
Building Lists from Pairs



By chaining another pair onto the previously constructed list, we've added 2 to the front of the list.
The resultant list is `'(2 1)`, a.k.a. `(list 2 1)`, or more literally `(cons 2 (cons 1 '()))`.

Building Nested Lists

So far we've only ever pointed the *cdr* pointer of a pair to another pair. But, we can point the *car* pointer to a pair to create a nested list. Notice that *every* chain of *cdr*'s ends in a pair of empty pointers. We can theoretically grow these nested pair connections arbitrarily (but finitely) deep and wide.

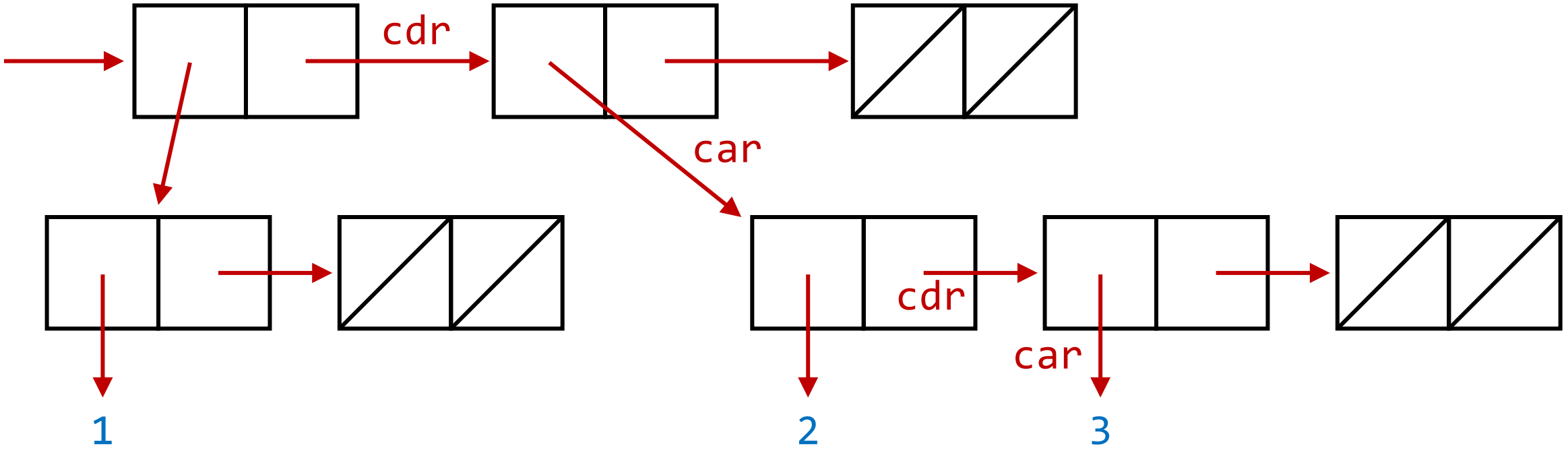


The diagram above represents the nested list `'((1) (2 3))`, or literally

```
(cons (cons 1 '())
      (cons (cons 2 (cons 3 '()))
            '())).
```

Building Nested Lists

To access a nested item or pair in the list, we need merely follow the chain of `car`'s and `cdr`'s **from** the desired target **to** the start of the list. Suppose in this list `'((1) (2 3))`, we want to access the `3`.



So to access the `3`, we would call the functions in the **reverse** of the order that we traversed them:
`(car (cdr (car (cdr '((1) (2 3)) ))))` yields `3`.
In Racket we can collapse up to 4 `car`/`cdr` calls into one, so the above call becomes `(cadadr '((1) (2 3)) )`.

An Aside on Subdefinitions

When writing more complicated functions, it can be useful to create helper functions to reduce redundant code. In Racket, you can begin the body of any function definition with a sequence of `define` expressions. These "subdefinitions" are not visible to outside functions, and can use the parameters of their parent functions.

Uglier code:

```
(define (f a b c x)
  (* (+ x (* 2 a))
     (+ x (* 2 b))
     (+ x (* 2 c)) ))
```

Cleaner code:

```
(define (f a b c x)
  (define (g z) (+ x (* 2 z)) )
  (* (g a) (g b) (g c) ))
```

We can use this technique to compute values as well, not just for cleanliness but for efficiency. It's better to compute something once and use the result twice than to compute it twice!

Less efficient code:

```
(define (g x)
  (* (+ x 1) (+ x 1) (+ x 1)
     (+ x 2) (+ x 2) (+ x 2) ))
```

More efficient code:

```
(define (g x)
  (define y (+ x 1))
  (define z (+ y 1))
  (* y y y z z z) )
```

Using subdefinitions is not required in the programming assignments; it's merely a strategy which can come in handy.