Exercise 1.30. The `sum` procedure above generates a linear recursion. The procedure can be rewritten so that the sum is performed iteratively. Show how to do this by filling in the missing expressions in the following definition:

```
(define (sum term a next b)
  (define (iter a result)
    (if <??>
        <??>
        (iter <??> <??>)))
  (iter <??> <??>))
```

Answer. To solve this problem, just remember the general strategy we have learnt to design a iterative procedure:

- Figure out a way to accumulate partial answer.
- Write our a table to analyze precisely:
  - initialization of first row
  - update rules for other rows
  - how to know when to stop
- Translate rules into Scheme code.

Let's think about how you might do the summation by hand. An advisable way would be drawing a table with several columns, each stands for a variable we need to keep track of on every step of computation, say, the current index(`a`), the upper bound(`b`) and the result. For example, for `(sum-cubes 1 5)`, you would immediately perform your computation in a form like Table 1.

a	result	b
1	0	5
2	1	5
3	9	5
4	36	5
5	100	5
6	225	5

Table 1. Raw Table for Evaluating `(sum-cube 1 5)`

The interface between procedure `sum` and `iter` above suggests that we should only keep track of 2 things on each step of our computation, although 3 variables are actually demanded. After investegating the structure of procedure `sum`, one might notice that by using block structure, we can probably keep the third variable `b` which set the upper bound outside procedure `iter`. Therefore, our original table evolves into a more concise version, as is shown in Table 2.

a	result
1	0
2	1
3	9
4	36
5	100
6	225

Table 2. An Updated Table for Evaluating `(sum-cube 1 5)`

Now, what we need to do is come up with a rule for how we are going to change the values in the table. In particular, to get the next value for result. As is shown in Table 3, considering this in term of procedure `sum-cube`, we take the the current value of `a`, evaluate its cube, then add that intermediate value to the current value of `result`, and that becomes the next entry in the `result` column. Similarly, we can easily capture the rule for getting the next value for `a`—just increase `a` by 1.

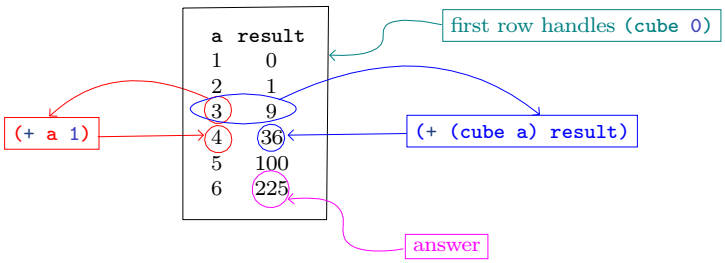

Table 3. The Schema of Process Evolution in Evaluating `(sum-cube 1 5)`

And how do we know when we are ready to stop? Well, that's easy. The last row we want in the table is the one when `a` is bigger than `b`.

Having done the analysis above, we are now able to write down our `sum-cubes` procedure in a iterative way:

```
(define (sum-cubes cube a inc b)
  (define (iter a result)
    (if (> a b)
        result
        (iter (inc a) (+ (cube a) result))))
  (iter a 0))
```

By following its prototype in the problem and generalizing our `sum-cubes` above, we can easily complete the `sum` procedure:

```
(define (sum term a next b)
  (define (iter a result)
    (if (> a b)
        result
        (iter (next a) (+ (term a) result))))
  (iter a 0))
```