Exercise 2.20.

The procedures `+`, `*`, and `list` take arbitrary numbers of arguments. One way to define such procedures is to use `define` with *dotted-tail notation*. In a procedure definition, a parameter list that has a dot before the last parameter name indicates that, when the procedure is called, the initial parameters (if any) will have as values the initial arguments, as usual, but the final parameter's value will be a *list* of any remaining arguments. For instance, given the definition

```
(define (f x y . z) <body>)
```

the procedure `f` can be called with two or more arguments. If we evaluate

```
(f 1 2 3 4 5 6)
```

then in the body of `f`, `x` will be `1`, `y` will be `2`, and `z` will be the list `(3 4 5 6)`. Given the definition

```
(define (g . w) <body>)
```

the procedure `g` can be called with zero or more arguments. If we evaluate

```
(g 1 2 3 4 5 6)
```

then in the body of `g`, `w` will be the list `(1 2 3 4 5 6)`.[1]

Use this notation to write a procedure `same-parity` that takes one or more integers and returns a list of all the arguments that have the same even-odd parity as the first argument. For example,

```
(same-parity 1 2 3 4 5 6 7)
(1 3 5 7)

(same-parity 2 3 4 5 6 7)
(2 4 6)
```

Answer.

Note the list we try to obtain has a property that all the subsequent elements of it share the same even-odd parity with the first argument. This indicates that the first argument of the sequence should be separated from the rest. Relate to the dotted-tail notation, all the subsequent elements should be enclose in a list. Hence, the steps to obtain a sublist that meets our need from the sequence

$$x\,.\,(y_1, y_2, \ldots, y_n)$$

would be:

- Identify the parity of the first argument x.
- Compose a sublist of $(y_1, y_2, \ldots, y_n)$ whose elements share the same parity with x.
- `Cons` the x on to the sublist we just obtained.

Now, we can express this idea as the procedure `same-parity`:

```
(define (same-parity x . y)
  (cons x (build-sp-sublist x y)))

(define (build-sp-sublist x items)
  (if (odd? x)
```


1. To define `f` and `g` using `lambda` we would write

```
(define f (lambda (x y . z) <body>))
(define g (lambda w <body>))
```

```
      (parity-sublist odd? items)
      (parity-sublist even? items)))

(define (parity-sublist parity? items)
  (cond ((null? items) '())
        ((parity? (car items))
         (cons (car items)
               (parity-sublist parity? (cdr items))))
        (else
          (parity-sublist parity? (cdr items)))))
```