The missing procedure is `(lambda (x) (append (list car s)) x)`. The way the procedure `subsets` works is by creating a tree where the left branch of each subtree will include all subsets of `s` without `(car s)` and the right branch of each subtree will include all subsets of `s` with `(car s)`. Besides the exclusion of a final step to append `(car s)` to each subset (using `map`) in the right branch, the two branches evolve identically, and each recursive call creates a new subtree where this behavior is observed. This occurs until the final `cdr` (`nil`) is reached, from which point lists of `subsets` are built by `map` procedures and then appended together.

```
(subsets (list 1 2 3))
(append (subsets (list 2 3))
        (map
          (lambda (x) (append (list 1) x))
          (subsets (list 2 3))))
(append
  (append (subsets (list 3))
          (map
            (lambda (x) (append (list 2) x))
            (subsets (list 3))))
  (map
    (lambda (x) (append (list 1) x))
    (append (subsets (list 3))
            (map
              (lambda (x) (append (list 2) x))
              (subsets (list 3))))))
(append
  (append
    (append (subsets nil)
            (map
              (lambda (x) (append (list 3) x))
              (subsets nil)))
    (map
      (lambda (x) (append (list 2) x))
      (append (subsets nil)
              (map
                (lambda (x) (append (list 3) x))
                (subsets nil)))))
  (map
    (lambda (x) (append (list 1) x))
    (append
      (append (subsets nil)
              (map
                (lambda (x) (append (list 3) x))
                (subsets nil)))
      (map
        (lambda (x) (append (list 2) x))
        (append (subsets nil)
                (map
                  (lambda (x) (append (list 3) x))
                  (subsets nil)))))))
(append
  (append
    (append (list nil)
            (map
              (lambda (x) (append (list 3) x))
              (list nil)))
    (map
      (lambda (x) (append (list 2) x))
      (append (list nil)
              (map
                (lambda (x) (append (list 3) x))
                (list nil)))))
  (map
    (lambda (x) (append (list 1) x))
    (append
      (append (list nil)
              (map
                (lambda (x) (append (list 3) x))
                (list nil)))
      (map
        (lambda (x) (append (list 2) x))
        (append (list nil)
                (map
                  (lambda (x) (append (list 3) x))
                  (list nil)))))))
(append
  (append
    (append (list nil)
            (list (list 3)))
    (map
      (lambda (x) (append (list 2) x))
      (append (list nil)
              (list (list 3)))))
  (map
    (lambda (x) (append (list 1) x))
    (append
      (append (list nil)
              (list (list 3)))
      (map
        (lambda (x) (append (list 2) x))
        (append (list nil)
                (list (list 3)))))))
(append
  (append
    (list nil (list 3))
    (map
      (lambda (x) (append (list 2) x))
      (list nil (list 3))))
  (map
    (lambda (x) (append (list 1) x))
    (append
      (list nil (list 3))
      (map
        (lambda (x) (append (list 2) x))
        (list nil (list 3))))))
(append
  (append
    (list nil (list 3))
    (list (list 2) (list 2 3)))
  (map
    (lambda (x) (append (list 1) x))
    (append
      (list nil (list 3))
      (list (list 2) (list 2 3)))))
(append
  (list nil (list 3) (list 2) (list 2 3))
  (map
    (lambda (x) (append (list 1) x))
    (list nil (list 3) (list 2) (list 2 3))))
(append
  (list nil (list 3) (list 2) (list 2 3))
  (list (list 1) (list 1 3) (list 1 2) (list 1 2 3)))
(list nil (list 3) (list 2) (list 2 3) (list 1) (list 1 3) (list 1 2) (list 1 2 3))
```

(() (3) (2) (2 3) (1) (1 3) (1 2) (1 2 3))

Here are the specifics I wrote out to help evaluate some of the more challenging reductions from above.

First we have to evaluate this code block:

```
(map
  (lambda (x) (append (list 3) x))
  (list nil)))
```

`(list nil)` is a pair where `car` and `cdr` are `nil`. This means passing `(list nil)` to `map` will make a new pair where the `car` is the procedure applied to `nil` (so appending `nil` to `(list 3)` produces `(list 3)`) and the `cdr` is the result of applying `map` to the `cdr` (which is `nil`, thus producing `nil`). Therefore the new pair has a `car` of `(list 3)` and a `cdr` of `nil`, which means it can be represented as `(list (list 3))`, or *((3))* since it is a list whose `car` points to *(3)*.

The next reduction is appending `(list nil)` to `(list (list 3))` which yields `(list nil (list 3))`:

```
(cons (car (list nil)) (append (cdr (list nil)) (list (list 3))))
(cons nil (append nil (list (list 3))))
(cons nil (list (list 3)))
```

Where `car` of the list is `nil` and `cadr` is a list with one value, 3. This is equivalent to a pair where the `car` is `nil` and the `cdr` is a list whose `car` is a list whose `car` is 3. Therefore, `(caadr (list nil (list 3))` is equivalent to `(caadr (cons nil (list (list 3))))`.

Now we have to deal with:

```
(map
  (lambda (x) (append (list 2) x))
  (list nil (list 3)))
```

`car` of `(list nil (list 3))` is `nil` and `cdr` is `(list 3)`

```
(cons (append (list 2) nil)
      (map proc (list 3)))
```

The latter, `(append (list 2) (list 3))`, will evaluate to `(list 2 3)`, so we get a pair where `car` is `(list 2)` and `cadr` is `(list 2 3)`. This means this all reduces to `(list (list 2) (list 2 3))` or *((2) (2 3))*.

Below is some of the remaining code from my original substitution that helped me reverse engineer the correct procedure for `map`. I noticed that everything reduced to `nil` and there needed to be a way to build out sets with the original numbers, which meant that at each recursion the procedure had to change. Using the `car` made sense, because it was what was disappearing from the rest of the structure, and needed to be added back in.

```
(append
  (append (subsets (3)) (map ? (subsets (3))))
  (map ? (append (subsets (3)) (map ? (subsets (3))))))
(append
  (append
    (append (subsets ()) (map ? (subsets ())))
    (map ? (append (subsets ()) (map ? (subsets ())))))
  (map ?
       (append
         (append (subsets ()) (map ? (subsets ())))
         (map ?
              (append (subsets ()) (map ? (subsets ())))))))
(append
  (append
    (append () (map ? ()))
    (map ? (append () (map ? ()))))
  (map ?
       (append
         (append () (map ? ()))
         (map ? (append () (map ? ()))))))
(append
  (append
    (append () (map ? ()))
    (map ? (append () (map ? ()))))
  (append (list 1)
       (append
         (append () (map ? ()))
         (map ? (append () (map ? ()))))))
```