Destructive programming and iteration

Destructive programming with set!

The special form set! changes the value of a local or global symbol destructively, i.e., its previous value is permanently lost.

(set! ⟨name⟩ ⟨newvalue⟩)

The name is not evaluated (otherwise it would be replaced by its current value), but its binding is changed to ⟨newvalue⟩, e.g.,

```
> (define x 4)
> x
4
> (set! x 5)
> x
5
```

So far we’ve been able to do a great deal without a need for destructive programming – and I recommend you avoid destructive programming when it’s not needed.

However, it’s a natural part of writing iterative procedures and working with vectors, which we’ll consider next.
Iteration

So far when we’ve wanted to do a similar thing multiple times, we’ve used recursion or higher-order procedures.

Recursion is all we need; any computable function can be written using recursion as the only control structure for repetition.

However, it can be easier to think about some problems in terms of a section of code that’s repeated during execution, changing the values of variables as it executes. This is called iteration.

while

The while statement repeats its code while a specified condition is true. Its format is:

\[(\text{while } \langle \text{condition} \rangle \langle \text{code} \rangle)\]

The \text{condition} is tested first; if it’s true, then the code (a series of expressions) is executed and the operation is repeated. In other words, keep going until \text{condition} is false.

The return value of a while statement is \(\text{(void)}\).

Steps:

1. Define any local variables you need before you start the while loop (how many you need depends on problem) using a \text{let} special form.

2. Start the while loop with a conditional expression that is initially true. This variable must eventually become false in order for the loop to end.

3. Use \text{set!} statements inside the while loop to change the value of variables and get closer to the case where the conditional statement returns false.

There are several kinds of iteration constructs we can use for iteration:

while
dotimes
dolist
Example: count
Let’s write a basic procedure that counts up to its input $n$:

```
> (count 5)
counter = 0
counter = 1
counter = 2
counter = 3
counter = 4
5
```

Example: recursive count

```
define count
(lambda (x)
  (letrec ((helper
              (lambda (count)
                (cond ((>= count x)
                        count)
                      (else
                        (printf "counter = ~A~%n" count)
                        (helper (+ count 1)))))))
    (helper 0)))
> (count 5)
counter = 0
counter = 1
counter = 2
counter = 3
counter = 4
5
```

Example: iterative count

```
declare count
(lambda (x)
  (let ((counter 0))
    (while (< counter x)
      (printf "counter = ~A~%n" counter)
      (set! counter (+ counter 1)))
    counter))
> (count 5)
counter = 0
counter = 1
counter = 2
counter = 3
counter = 4
5
```

Example: sum-to-n

Let’s write a procedure that adds up the numbers from 0 to its input, $n$, printing the equation before returning the sum:

```
> (sum-to-n 10)
0 + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 = 55
```
Example: recursive \texttt{sum-to-n}

(\texttt{define sum-to-n} (lambda (n))
 \texttt{(letrec \{(sum-rec (lambda (count sum)
 \texttt{(cond \{ (= count n)
 \texttt{(printf \"~A = \" count)
 (+ count sum))
 \texttt{else}
 \texttt{(printf \"~A + \" count)
 (sum-rec (+ count 1)
 (+ sum count))\})\}))
 (sum-rec 0 0)))}

Example: iterative \texttt{sum-to-n}

Without worrying about the \texttt{printf} statements:

(\texttt{define sum-to-n} (lambda (n))
 \texttt{(let ((counter 0)
 \texttt{(sum 0))
 \texttt{(while (<= counter n)
 \texttt{(set! sum (+ sum counter))
 \texttt{(set! counter (+ counter 1))))
 \texttt{sum})))}

Example: iterative \texttt{sum-to-n}

With \texttt{printf} statements:

(\texttt{define sum-to-n} (lambda (n))
 \texttt{(let ((counter 0)
 \texttt{(sum 0))
 \texttt{(while (<= counter n)
 \texttt{(if (< counter n)
 \texttt{(printf \"~A + \" counter))
 \texttt{(if (= counter n)
 \texttt{(printf \"~A = \" counter))
 \texttt{(set! sum (+ sum counter))
 \texttt{(set! counter (+ counter 1))}
 \texttt{sum}))})

Design pattern for a loop that looks for something

(\texttt{let ((done #f))
 \texttt{(while (not done)
 \texttt{...}
 \texttt{(if condition
 \texttt{(set! done #t))}
 \texttt{...}))}
(define index-of-num!
  (lambda (lon num)
    (let ((index 0) ;; position of number in the list
           (answer #f) ;; no such number in list
           (done #f)) ;; loop control variable
      (while (not done)
        (cond
          ;; Base case 1: list is empty and num not found;
          ;; stop while loop
          ((empty? lon)
           (set! done #t))
          ;; Base case 2: found num in list; return index
          ((= num (first lon))
           (set! answer index)
           (set! done #t))
          ;; Recursive case
          (else
           (set! index (+ index 1))
           (set! lon (rest lon)))
        answer)))

> (index-of-num! '(0 2 3 4 5 6 7 8 9 10) 7)
6

The general form of a **dotimes** statement is

```
(dotimes (var n) code)
```

The variable `var` is bound inside the **dotimes** as if it appeared in a `let`. The value of `var` is set to 0 and the `code` (a series of expressions) is executed; then the value of `var` is incremented by 1 and the `code` is repeated, and so on.

The total number of executions is `n`, so the last value of `var` is `n-1`. If `n` is 0 or negative, the `code` is not executed.

The return value of a **dotimes** statement is `(void)`.

---

**dotimes**

Another special form for writing loops is **dotimes**, which does something, `n` times, e.g.,  > (dotimes (i 15) 
    (printf "~A " i))
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

This is equivalent to a **while** statement, but we don’t need to explicitly increase the variable `i` by 1 each time:

  > (while (< i 15) 
    (printf "~A " i)
    (set! i (+ i 1)))

We can use **dotimes** to define an iterative factorial procedure:

```
(define factorial
  (lambda (n)
    (let ((product 1))
      (dotimes (i n)
        (set! product
          (* (+ i 1) product))
      product)))))
```
**dolist**

When we want to iterate through the items in a list, we can use the `dolist` statement:

```
(dolist (var lst) code)
```

The variable `var` is bound inside the `dolist`. The value of `var` is set to successive elements of the list `lst` and the `code` (a series of expressions) is executed for each of those elements. If `lst` is empty, the code is not executed.

Using a `while` loop to walk through a list:

```
(let ((listy '(a b c))
    (elt #f))
  (while (not (null? listy))
    (set! elt (first listy))
    (printf "elt: ~A~%" elt)
    (set! listy (rest listy))))
```

Equivalent (and easier) `dolist` version:

```
(dolist (elt '(a b c))
  (printf "elt: ~A~%" elt))
```

We can use the `dolist` special form to compute the sum of the elements of a list:

```
(define sum-list
  (lambda (lst)
    (let ((sum 0))
      (dolist (elt lst)
        (set! sum (+ sum elt)))
      sum)))
```

```
> (sum-list '(1 4 5 7))
17
```

---

**Acknowledgments**

This lecture incorporates material from:

Simon Ellis
Gordon S. Novak Jr
Jennifer Walter