

Abstract Data Types and Stacks

Contents:

- Abstract Data Types
- Stacks as an Abstract Data Type
- Stacks in Scheme
- Controlling Recursion

Abstract Data Types

An *abstract data type* is a set of values and operations on those values, considered independently of the ways in which those values might be represented and the operations implemented in actual programs. Separating the definition of an abstract data type from its implementation is a technique that has been found to be especially useful in the development of large software systems.

Objects produced by a constructor procedure, such as the switches in the reading on objects in Scheme, can often be developed efficiently from the definition of an abstract data type, with the advantage that other procedures that take such objects as arguments cannot operate on them or modify them in any way not considered by the definition of the abstract data type. As a result, the methods of such an object can often be implemented in such a way as to preserve certain simplifying *invariants* -- conditions that are known to be true at the beginning and end of the execution of each method. Relying on such invariants often allows the programmer to dispense with some precondition tests in the methods, because the invariants imply that the preconditions will be met whenever the method is called.

As illustrations of the use of abstract data types in the development of programs, we consider a frequently encountered ADT that Scheme happens not to supply: the *stack*.

Stacks as an Abstract Data Type

Conceptually, the stack abstract data type mimics the information kept in a pile on a desk. Informally, we first consider materials on a desk, where we may keep separate stacks for bills that need paying, magazines that we plan to read, and notes we have taken. We can perform several operations that involve a stack:

- Start a new stack.
- Place new information on the top of a stack,
- Take the top item off of the stack.
- Read the item on the top without removing it.
- Determine whether a stack is empty. (On very rare occasions, there may be nothing at the spot normally occupied by the stack.)

These operations allow us to do all the normal processing of data at a desk. For example, when we receive bills in the mail, we add them to the pile of bills until payday comes. We then take the bills, one at a time, from the top of the pile and pay them until the money runs out.

When discussing these operations, it is conventional to call the addition of an item to the top of the stack a *push* operation and the deletion of an item from the top a *pop* operation. (These terms are derived from the workings of a spring-loaded rack containing a stack of cafeteria trays. Such a rack is loaded by pushing the trays down onto the springs; as each diner removes a tray, the lessened weight on the springs causes the stack to pop up slightly.)

Here is a more formal definition of the stack ADT: A stack is a data structure containing zero or more elements, on which the following operations can be performed:

- *create* Create a new, empty stack object.
- *empty* Determine whether the stack is empty; return `#t` if it is and `false` if it is not.
- *push* Add a new element at the top of a stack.
- *pop* Remove an element from the top of the stack and return it. (This operation has a precondition: It cannot be performed if the stack is empty.)
- *top* Return the element at the top of the stack (without removing it from the stack). (This operation, too, can be performed only if the stack is not empty.)

This abstract data type definition says nothing about how we will program the various stack operations; rather, it tells us how stacks can be used. We can infer some limitations on how we can use the data. For example, stack operations allow us to work with only the top item on the stack. We cannot look at elements farther down in the stack without first using *pop* operations to clear away items above the desired one.

A *push* operation always puts the new item on top of the stack, and this is the first item returned by a *pop* operation. Thus, the last piece of data added to the stack will be the first item removed.

Stacks in Scheme

We can implement stacks in Scheme as objects that respond to the messages `':empty?`, `':push!`, `':pop!`, and `':top`. The *create* operation will correspond to the constructor procedure `make-stack`, which takes no arguments and returns an empty stack.

But how do we store the values in the stack? We use a list. The front element of the list represents the top of the stack. To push something on the stack, we cons it at the front of the list. To pop the stack, we take the `cdr` of the list. Since we want to change the stack, we keep a reference to the list in a vector, which we call `stk`.

```
;;; Procedure:
;;;   make-stack
;;; Parameters:
;;;   (none)
;;; Purpose:
;;;   Creates a stack
;;; Produces:
```

```

;;; stack, an object
;;; Postconditions:
;;; stack responds to the following messages:
;;; :type
;;; Returns the type of the object.
;;; :->string
;;; Summarizes the stack in a string. (Does not show contents.)
;;; :empty?
;;; Check if the stack is empty.
;;; :push! value
;;; Push a value on the stack.
;;; :pop!
;;; Get the top value on the stack and remove it.
;;; :top
;;; Get the top value on the stack and do not remove it.
(define make-stack
  (lambda ()
    (let ((stk (vector null)))
      (lambda (message . arguments)
        (let ((lst (vector-ref stk 0)))
          (cond ((eq? message ':type)
                 'stack)

                ; The to-string message intentionally reveals little
                ; about the contents.
                ((eq? message ':->string)
                 "#<stack>")

                ((eq? message ':empty?)
                 (null? (vector-ref stk 0)))

                ((eq? message ':push!)
                 (if (null? arguments)
                     (error "stack:push!: an argument is required")
                     (vector-set! stk 0
                                   (cons (car arguments) lst))))

                ((eq? message ':pop!)
                 (if (null? lst)
                     (error "stack:pop!: the stack is empty")
                     (begin
                      (vector-set! stk 0 (cdr lst))
                      (car lst))))

                ((eq? message ':top)
                 (if (null? lst)
                     (error "stack:top: the stack is empty")
                     (car lst)))

                (else
                 (error "stack: unrecognized message"))))))))

```

Since the vector `stk` is allocated during the definition process, outside of the `lambda`-expression for the procedure being returned, it will persist as part of the object between operations on that object. Further, note that a different static variable is created each time `make-stack` is invoked. Thus, a program can arrange for the construction of any number of stacks, which can be pushed and popped independently.

Controlling Recursion

Stacks are also useful when dealing with more complex forms of recursion, such as recursive procedures that call themselves multiple times. Rather than including both recursive calls in our code, we put information about the recursive call on the stack, and then repeatedly process the remaining values on the stack.

For example, suppose we have a tree of values and want to simultaneously count the number of symbols, strings, and numbers that appear in the tree. We'll return a list of three values (number of symbols, number of strings, number of numbers). We'll call that procedure `tally-things`.

Because writing procedures that recurse on trees can be difficult, let's start by writing such a procedure for lists.

The base case is easy: For an empty list, there are no symbols, strings, or numbers, so we return a list of three 0's.

```
(cond
  ((null? lst) (list 0 0 0))
```

Now, let's suppose we see a useful value, such as a symbol. What do we want to do? We want to recurse on the rest of the list, and then add 1 to the car. Here's one way to do so.

```
((symbol? (car lst))
 (cons (+ 1 (car (tally-things (cdr lst))))
       (cdr (tally-things (cdr lst)))))
```

I don't know if your mental alarms are going off yet, but they should be. The two identical recursive calls here can lead to a lot of extra work. Hence, we should do a single recursive call and name the result.

```
((symbol? (car lst))
 (let ((recursive-result (tally-things (cdr lst))))
   (cons (+ 1 (car recursive-result))
         (cdr recursive-result))))
```

Our code for strings is similar, except that we're filling in the middle value.

```
((string? (car lst))
 (let ((recursive-result (tally-things (cdr lst))))
   (list (car recursive-result)
         (+ 1 (cadr recursive-result))
         (caddr recursive-result))))
```

Putting it all together, we get something like the following:

```
;;; Procedure:
;;;  tally-things
;;; Parameters:
;;;  lst, a list.
;;; Purpose:
;;;  Count the numbers of symbols, strings, and numbers that
;;;  appear in lst.
```

```

;;; Produces:
;;; (symbol-tally string-tally number-tally), a list of three integers
;;; Preconditions:
;;; (None)
;;; Postconditions:
;;; symbol-tally contains the number of symbols that appear at the
;;; top level of lst.
;;; string-tally contains the number of strings that appear at the
;;; top level of lst.
;;; number-tally contains the number of numbers that appear at the
;;; top level of lst.
(define tally-things
  (lambda (lst)
    (cond
      ((null? lst) (list 0 0 0))
      ((symbol? (car lst))
       (let ((recursive-result (tally-things (cdr lst))))
         (cons (+ 1 (car recursive-result))
               (cdr recursive-result))))
      ((string? (car lst))
       (let ((recursive-result (tally-things (cdr lst))))
         (list (car recursive-result)
               (+ 1 (cadr recursive-result))
               (caddr recursive-result))))
      ((number? (car lst))
       (let ((recursive-result (tally-things (cdr lst))))
         (list (car recursive-result)
               (cadr recursive-result)
               (+ 1 (caddr recursive-result)))))
      (else recursive-result))))

```

Of course, it's a bit of a pain that we construct lists of values in each recursive call only to immediately deconstruct each list in the surrounding call. The normal solution is to write a kernel that accumulates the three tallies as we go.

```

(define tally-things
  (lambda (lst)
    (let kernel ((remaining lst)
                 (symbol-tally 0)
                 (string-tally 0)
                 (number-tally 0))
      (cond
        ((null? remaining) (list symbol-tally string-tally number-tally))
        ((symbol? (car remaining))
         (kernel (cdr remaining) (+ 1 symbol-tally) string-tally number-tally))
        ((string? (car remaining))
         (kernel (cdr remaining) symbol-tally (+ 1 string-tally) number-tally))
        ((number? (car remaining))
         (kernel (cdr remaining) symbol-tally string-tally (+ 1 number-tally)))
        (else
         (kernel (cdr remaining) symbol-tally string-tally number-tally)))))

```

This version is both more efficient than the previous version and easier to read (at least for many programmers).

Now, let's turn to the problem of tallying in a tree. (Remember, that's where we began this problem.) The normal strategy for trees is to have a case for pairs, for the empty list, and for other values (the leaves). As in the case of lists, let's start by writing a version in which we don't have a kernel that keeps track of the various tallies. In fact, this version is cleaner than the original version for lists.

For the case `null`, we're back to return a list of three 0's.

```
(cond
  ((null? tree) (list 0 0 0))
```

When we hit a leaf, we return the appropriate list of values.

```
((symbol? tree) (list 1 0 0))
((string? tree) (list 0 1 0))
((number? tree) (list 0 0 1))
```

Now, pairs are the hard part. In this case, we need to count in both subtrees and then add.

```
((pair? tree)
  (let ((left (tally-things (car tree)))
        (right (tally-things (cdr tree))))
    (list (+ (car left) (car right))
          (+ (cadr left) (cadr right))
          (+ (caddr left) (caddr right)))))
```

Putting it all together, we get the following:

```
;;; Procedure:
;;; tally-things
;;; Parameters:
;;; tree, a tree
;;; Purpose:
;;; Count the numbers of symbols, strings, and numbers that
;;; appear in the tree.
;;; Produces:
;;; (symbol-tally string-tally number-tally), a list of three integers
;;; Preconditions:
;;; (None)
;;; Postconditions:
;;; symbol-tally contains the number of symbols that appear in tree.
;;; string-tally contains the number of strings that appear in tree.
;;; number-tally contains the number of numbers that appear in tree.
(define tally-things
  (lambda (lst)
    (cond
      ((null? tree) (list 0 0 0))
      ((symbol? tree) (list 1 0 0))
      ((string? tree) (list 0 1 0))
      ((number? tree) (list 0 0 1))
      ((pair? tree)
       (let ((left (tally-things (car tree)))
             (right (tally-things (cdr tree))))
         (list (+ (car left) (car right))
               (+ (cadr left) (cadr right))
               (+ (caddr left) (caddr right))))))
```

```

      (list (+ (car left) (car right))
            (+ (cadr left) (cadr right))
            (+ (caddr left) (caddr right))))
    (else (list 0 0 0))))

```

All well and good, except for one little thing: Once again, we're building lots and lots of lists, only to take them apart. What do we do? Well, we want to add the tallies we added before. Unfortunately, we have the little problem of two recursive calls to `tally-things`. So, what do we do? We get to the reason that this example appears in this reading: We can use a stack to keep track of the parts of the tree we have not yet processed. At each step, we process the next remaining part of the tree. If it's a symbol, string, or number, we increment the appropriate tally. If it's a pair, we push both halves on the stack for future processing.

```

(define tally-things
  (lambda (tree)
    (let ((remaining (make-stack)))
      (remaining ':push! tree)
      (let kernel ((symbol-tally 0)
                   (string-tally 0)
                   (number-tally 0))
        (if (remaining ':empty?)
            (list symbol-tally string-tally number-tally)
            (let ((current (remaining ':pop!)))
              (cond
               ((null? current)
                (kernel symbol-tally string-tally number-tally))
               ((symbol? current)
                (kernel (+ 1 symbol-tally) string-tally number-tally))
               ((string? current)
                (kernel symbol-tally (+ 1 string-tally) number-tally))
               ((number? current)
                (kernel symbol-tally string-tally (+ 1 number-tally)))
               ((pair? current)
                (remaining ':push! (cdr current))
                (remaining ':push! (car current))
                (kernel symbol-tally string-tally number-tally))
               (else
                (kernel symbol-tally string-tally number-tally))))))))))

```

A bit longer, but significantly more efficient.

The idea of storing parts of a tree (or otherwise storing parts of a multiply-recursive procedure) on a stack is a useful programming strategy. Keep an eye out for other occasions to use it.

Copyright © 2006 Samuel A. Rebelsky. This work is licensed under a Creative Commons Attribution-NonCommercial 2.5 License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc/2.5/> or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.