

Stacks

Summary: We explore some simple applications of stacks.

Contents:

- Preparation
- Exercises
 - Exercise 1: Matching Tags
 - Exercise 2: The Size of Stacks
 - Exercise 3: Purging Stacks
 - Exercise 4: Validating Numbers of Parameters
- For Those With Extra Time

Preparation

- Start DrScheme.
- Make a copy of the the code for the stack object constructor from the reading on stacks.

Exercises

Exercise 1: Matching Tags

Documents on the World Wide Web usually contain special strings, called *tags*, that serve as instructions to the browser about what the document contains, how it is structured, and how the text should be displayed. In many cases, tags occur in pairs: The opening tag marks the beginning of a region of text that constitutes some natural unit within the document structure or should be displayed in some special way, and closing tag marks the end of that region.

Netscape and other browsers use a stack of tags like this one -- a stack containing tags that must eventually be matched but have not been matched yet -- to determine whether the HTML document to be displayed is correctly constructed. Write a Scheme procedure `correctly-nested?` that takes a list of HTML opening and closing tags and determines whether they are correctly nested.

You need not handle singleton tags.

```
> (correctly-nested? (list "<html>" "<head>" "<title>" "</title>"
                        "</head>" "<body>" "<b>" "</b>" "</body>" "</html>"))
#t
> (correctly-nested? (list "<html>" "<head>" "</html>" "</head>"))
#f
```

You may find the following procedures helpful.

```
;;; Procedure:
;;; string-index
;;; Parameters:
;;; str, a string
;;; ch, a character
;;; Purpose:
;;; Finds the first index of ch in str.
;;; Produces:
;;; index, an integer (or #f)
;;; Preconditions:
;;; (none)
;;; Postconditions:
;;; If ch appears in str, (string-ref str index) is ch and
;;; For all j < index, (string-ref str index) is not ch.
;;; If ch does not appear in str, ch is #f.
(define string-index
  (letrec ((kernel
            (lambda (str ch pos len)
              (cond
                ((= pos len) #f)
                ((eq? (string-ref str pos) ch) pos)
                (else (kernel str ch (+ pos 1) len))))))
    (lambda (str ch)
      (kernel str ch 0 (string-length str)))))

;;; Procedure:
;;; tag?
;;; Parameters:
;;; str, a string
;;; Purpose:
;;; Determine whether str seems to represent an HTML tag.
;;; Produces:
;;; is-tag?, a Boolean
;;; Preconditions:
;;; (none)
;;; Postconditions:
;;; If str starts with a less-than sign ("<"), ends with a
;;; greater-than sign (">"), and contains no other less-than
;;; such signs, then is-tag? is true.
;;; Otherwise, is-tag? is false.
;;; Problems:
;;; Although it is officially possible to include a less-than or
;;; greater-than sign in a tag, we consider such tags invalid.
(define tag?
  (lambda (str)
    (if (not (string? str)) #f
        (let* ((len (string-length str))
               (contents (substring str 1 (- len 1))))
          (and (>= len 2)
               (eq? (string-ref str 0) #\<)
               (eq? (string-ref str (- len 1)) #\>)
               (not (string-index contents #\<))
               (not (string-index contents #\>)))))))

;;; Procedure:
```

```

;;; end-tag?
;;; Parameters:
;;; str, a string
;;; Purpose:
;;; Determines whether str seems to represent an ending HTML tag.
;;; Produces:
;;; is-end-tag?, a Boolean.
;;; Problems:
;;; Although it is officially possible to include a less-than or
;;; greater-than sign in a tag, we consider such tags invalid.
(define end-tag?
  (lambda (str)
    (and (tag? str)
         (eq? (string-ref str 1) #\))))

;;; Procedure:
;;; tag-name
;;; Parameters:
;;; tag, a string
;;; Purpose:
;;; Extracts the name of the tag from the HTML.
;;; Produces:
;;; name, a string
;;; Preconditions:
;;; (tag? tag) must hold.
;;; Postconditions:
;;; name is the name of the tag.
;;; Problems:
;;; Currently supports only unparameterized tags.
(define tag-name
  (lambda (tag)
    (if (eq? (string-ref tag 1) #\/)
        (substring tag 2 (- (string-length tag) 1))
        (substring tag 1 (- (string-length tag) 1)))))

```

Exercise 2: The Size of Stacks

Some authors add another operation to the definition of the stack ADT: **size**, which returns the number of elements in the stack. Extend the Scheme implementation of `make-stack` above so that the stacks it constructs will accept the message `' :size` and perform this operation when it is received.

Exercise 3: Purging Stacks

Some authors add another operation to the definition of the stack ADT: **purge**, which eliminates all the elements in the stack. Extend the Scheme implementation of `make-stack` so that the stacks it constructs will accept the message `' :purge!` and perform the corresponding operation.

Exercise 4: Validating Numbers of Parameters

The current implementation of `make-stack` does not verify that the stack is called correctly. For example, one could provide another parameter to `' :pop!`, even though no such parameters should be permitted.

Extend the implementation of `make-stack` so that each operation checks that it is associated with the correct number of additional parameters.

For Those With Extra Time

As you may know, processing arithmetic expressions, either by hand or by computer, can lead to some confusions. For example, different infix calculators evaluate $3+4*5$ differently. (Interestingly, the Microsoft Calculator gives different results depending on whether you use the basic or scientific calculator.)

Scheme includes parentheses so that we don't have ambiguities about order of evaluation. However, the parentheses add their own complication, as you have no doubt noticed.

Is there a way to represent arithmetic expressions unambiguously, but without parentheses? If we restrict ourselves to fixed-arity operations, yes. For many years, HP calculators supported a notation called *Reverse Polish Notation* (RPN), so named because it's the opposite of Polish notation, which looks a lot like Scheme notation, but without the parentheses. In RPN, you put the operands before the operation. For example, to add 2 and 3, you write `2 3 +`.

It turns out to be very easy to process RPN expressions if you have a stack.

- If the next value in the expression is a number, push it on the stack.
- If the next value in the expression is a procedure, pop its parameters (traditionally, two parameters), apply the procedure, and then push the result.
- At the end of the expression, print the top of the stack.

Suppose we represent each RPN expression as a list of numbers and procedures. Here's the start of an evaluation procedure

```
(define rpn-eval
  (lambda (expression)
    (let ((stack (make-stack)))
      (let kernel ((remainder expression))
        (cond
          ((null? remainder)
           (stack ':top))
          ((number? (car remainder))
           ...
           (kernel (cdr remainder)))
          ((procedure? (car remainder))
           ...
           (kernel (cdr remainder)))
          (else
           (error "rpn-eval: invalid expression"))))))))
```

Fill in the ellipses to make evaluation work correctly.

```
> (rpn-eval (list 2 3 + 4 *))
20
> (rpn-eval (list 2 3 4 + *))
14
> (rpn-eval (list 2 5 -))
-3
```

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.