CSE1729 - Introduction to Programming November 17, 2020

Problem Set 9

1. Define a SCHEME function named `make-clock` which takes two parameters which represent the current time in hours and minutes. Your clock object should store the current time in minutes only. Your clock object should expose three functions for working with the clock object:

   - `'tick` - advances the time of the clock object by one minute
   - `'time` - returns a string with the current 12-hour time indicating whether the current time is AM or PM
   - `'military` - returns a string representing the current time in military (24-hour) time

```
>(define get-time (clock 'time))
>(define get-mil (clock 'military))
>(display (get-time))
>((clock 'tick))
>(display (get-time))
>((clock 'tick))
>(display (get-time))
>(display (get-mil))
9:00 PM
9:01 PM
9:02 PM
21:02
```

   Note: If the time is displayed in military time, your object should print any leading zeros for the hour. Both time formats should print leading zeroes for the minutes. For example:

```
>(define clock (make-clock 2 0))
>(define get-time (clock 'time))
>(define get-mil (clock 'military))
>(display (get-time))
>(display (get-mil))
>((clock 'tick))
>(display (get-time))
>(display (get-mil))
2:00 AM
02:00
2:01 AM
02:01
```

You can display the newline character in a string (e.g. `"\n"`) to move to the next line after displaying the time. You also may use the `string-append` function to build up strings from different components, and the `number->string` function to convert a number to a string representing that number.

2. STACK APPLICATIONS
   In this problem you will define an object that implements the *Stack* abstract data type *using a list to store the elements*. When the object is created, it starts as an empty stack.

   To implement the stack, the object will initially create an empty list, which it will store the elements of the stack. The stack contents are then maintained with the following convention: to represent the stack containing the elements $e_1, e_2, \ldots, e_n$ (where e_1 is the *top* of the stack and e_n is the *bottom* of the stack) the list will contain the elements as shown in Figure 1, just below. Observe that the bottom element of the stack is always at

   $$'(e_1\ e_2\ e_3 \ldots e_k\ )$$

   Figure 1: Layout of a stack in a list.

   the end of the list. The top element of the stack can be accessed at the front of the list. To place a new element on the top of the stack, one needs only add it to the front of the list. Popping an element off the top of the stack is handled by returning the appropriate element *(take particular care to ensure you are returning the proper value when using destructive assignment)* and "removing" that element from the front of the list. Your object should expose methods for

   empty? Returns a Boolean value (`\#t` or `\#f`) depending on whether the stack is empty.

   push Pushes a new element onto the top of the stack.

   pop Pops off the top element from the stack and returns it.

   top Returns the value of the top of the stack (without changing the contents of the stack).

   Thus, your object should have the form:

```
(define (make-stack)
  (let (...)                              ;; internal stack variables
    (define (empty?) ...)          ;; stack methods
    (define (push x) ...)
    (define (pop) ...)
    (define (top) ...)
    (define (dispatcher ...) ...) ;;the dispatcher
    dispatcher))
```

3. (EVALUATION OF POSTFIX EXPRESSIONS) Given a list of operands and operators that represent a postfix expression, one can use a stack to evaluate the postfix expression. Its true! The algorithm to do this is as follows:

Algorithm 1 Evaluate Postfix Expression

```
repeat
  if operand at front of input string then
    push operand onto stack
  else
    pop stack to remove operands
    apply operator to operand(s)
    push result onto stack
  end if
until input string is empty
```

For example, to evaluate the expression "23 15 +" we would:

1. Push 23 onto the stack
2. Push 15 onto the stack
3. "+" is an operator, so
   (a) Pop second operand *(note the operands are popped in reverse order)*
   (b) Pop first operand
   (c) Apply the operator (in this case, addition)
   (d) Push the result back onto the stack
4. Once the expression has been evaluated, the result is on the top of the stack.

Define a SCHEME function, named `(eval-postfix p)`, that will take a postfix expression (stored in a list of integers representing operands and characters representing operators), evaluate that expression, and return the result.
Your function should support the operations of addition (`#\+`), subtraction (`#\-`), multiplication (`#\*`), division(`#\/`), and exponentiation(`#\^`).
You may want to work incrementally by starting with a function that takes a character representing an operator and can pop two operands off of a stack, evaluate the operator and push the result back on the stack. Next you can add a function that evaluates a postfix expression by pushing operands onto the stack and evaluating operators when encountered. Note, you may want to use the `number?` function to determine whether an item in the list is an operand or an operator.

Traversing Trees without Recursion

4. One ordering for visiting nodes in a tree starts at the root node, visits the root, then the left subtree, and the right subtree. In visiting nodes in this order, the algorithm traverses the tree as far (deep) as it can go down the left side before backtracking and visiting the right subtree of the last node visited. This is referred to as a depth first search (DFS) traversal and is shown in Figure 2.

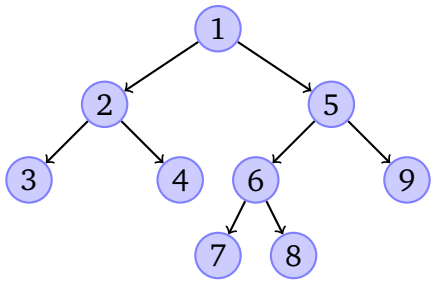

Figure 2: A binary tree showing the order in which nodes are visited in a DFS traversal as the value at each node.

One implementation of DFS traversal uses a Stack ADT to store nodes to be visited next. When nodes are visited, their right and left subtrees are pushed onto the stack to be visited later once the algorithm backtracks. Some implementations of this algorithm for graphs maintain a list of nodes that have been visited in case there are cycles in the graph. However, we are working explicitly with trees, which have no cycles. So, we can safely neglect to record which nodes have been visited. In order to traverse the tree in DFS order using a Stack, start at the root by pushing the root node onto an empty stack. We will build a list of the values at the nodes in the order visited. We do this by repeatedly popping a node from the stack, adding the value at that node to the front of a list whose tail is the list constructed by visiting the remaining nodes in the Stack, and then pushing the right subtree and then the left subtree onto the stack (if they are not the empty tree). Note the order of pushing subtrees onto the stack which ensures the left subtree will be traversed before the right subtree. See Algorithm 2. Eventually, you will have visited all nodes in the tree and the Stack will be empty. Define a SCHEME function, named (`dfs T`), that uses a Stack ADT object, created with an implementation of (`make-stack`), to produce a list of the values at nodes of the binary tree `T` in the order in which the nodes were visited in a DFS traversal of the tree.

Algorithm 2 DFS Traversal of a Tree

Require: A stack ADT containing the root of the tree to traverse
  repeat
    pop tree node from stack
    if right child of node is not null **then**
      push right child onto stack
    end if
    if left child of node is not null **then**
      push left child onto stack
    end if
    "visit" node
  until the stack is empty

5. Another traversal order visits a nodes children before visiting *their* children (i.e. the node's children's children). This traversal ordering is referred to as Breadth-First Search (BFS) order and is shown in Figure 3. Again, since we are working explicitly with binary trees, there are no cycles in our "graph" and we can safely neglect to record which nodes have been visited already by our algorithm.

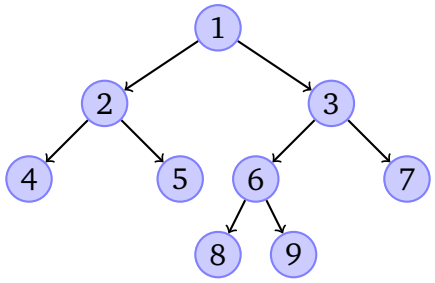

Figure 3: A binary tree showing the order in which nodes are visited in a BFS traversal as the value at each node.

To implement a function showing BFS traversal of a tree, we use a Queue ADT. In this case, the root node is enqueued into the Queue ADT. Next, a node is dequeued, that node's children are enqueued into the queue ADT, and finally the value at the dequeued node is added to the front of the list obtained by visiting the rest of the nodes in the queue. See Algorithm 3. Define a SCHEME procedure, named (`bfs T`) which traverses a binary tree in BFS order and produces a list of the values of the nodes of the tree in the order in which the nodes were "visited."

Algorithm 3 BFS Traversal of a Tree

Require: A queue ADT containing the root of the tree to traverse
  repeat
    dequeue tree node from the queue
    if left child of node is not null **then**
      enqueue left child onto queue
    end if
    if right child of node is not null **then**
      enqueue right child onto queue
    end if
    "visit" node
  until the queue is empty