rev 08/08/2016 jbo

Pseudocode Reference Sheet

Note: This document has been developed by ***WeTeach_CS***, and is solely based on current study materials and practice tests provided on the TEA website. It has been indicated that an official pseudocode reference document will be provided with the new test set to release in August 2016.

Data Types	
`int x <- 10`	An integer variable
`boolean isMoving <- true`	A Boolean (true or false) variable
`char grade <- 'A'`	A character variable
`string fruit <- "Apple"`	A string variable
`float num <- 3.14`	A floating point (decimal) variable

Operators, Assignment, Output and Return Statements	

`a ← expression`
`a <- expression`
`a ← b * 3 + 2`
`a <- b * 3 + 2`
	Evaluates `expression` and assigns the result to the variable, a right to left operation.

`print k`
`print (k)`
 	
Note: Both print styles are demonstrated in the practice test
	Prints the value of k

`return 1 + b`	Returns the sum of 1 and b

Arithmetic Operators

`a+b`
`"dog" + "cat"  -> "dogcat"`
`"dog " + "cat"  -> "dog cat"`
`"dog" + 5 + 1.2 -> "dog51.2"`
	The arithmetic operators `+, -, *, /` and `%` perform arithmetic operations on **`a`** and **`b`**, and work on both integer and floating point values.
	
	Concatenation of strings also uses the + operator
	
	For all arithmetic operations, when both operands are integers, the result is an integer. If either or both operands are of type float (a decimal), the result is a float.
	
	For example:
	4 + 5 results in 9
	4 + 5.0 results in 9.0
	3 / 2 is 1
	3.0 / 2 is 1.5

`a-b`	
`a*b`	
`a/b`	
`a%b`	

Note: The order of operations is the same as it is for mathematics: Multiplicative operations first (*, /, %) with multiple instances evaluated in order from left to right, and then additive operations (+,-), also evaluated in order from left to right.

Relational Operators

`a == b`	All of these relational operators are used to test the relationship between two variables, expressions, or values, with a boolean result of **`true`** or **`false`**. For example, **`a == b`** evaluates to true if a and b contain equal values, otherwise it evaluates to false. Here are other examples:
`a ≠ b, a!=b, a<>b`	
`a < b`	
`a > b`	

`a ≥ b, a>=b`	`4 == 5` is `false`
	`4 <= 5` is `true`
	`4 < 5` is `true`
	`4 >= 5` is `false`
	`4 > 5` is `false`
	`4 != 5` is `true`

`a ≤ b, a<=b`	

Note: There is no evidence of any distinction between primitives and objects in any of the practice test questions or study materials for the test, therefore it will be assumed that any comparison between strings, or any other object data types, will indeed be comparing the contents of the data, and not the memory locations.

Logical Operators

`NOT condition` `! condition`	Reverses the value of the boolean expression, becoming true if it is false, or false if it is true.
`condition1 AND condition2`	Evaluates to true if both condition1 and condition2 are true; otherwise evaluates to false.
`condition1 XOR condition2`	Evaluates to true if the conditions are opposite - true AND false, or false AND true If the conditions are the same, it evaluates to false.
`condition1 OR condition2`	Evaluates to true if either or both conditions are true, false if both are false.

Note: The order of precedence for the logical operators, should there be expressions with multiple operations, is as they are listed above: NOT, AND, XOR, then OR … **NAXO** for short.

Operator Precedence

The overall order of operator precedence for the certification exam pseudocode system is as follows:

1. Unary (positive `+`, negative `-`, `NOT !`)
2. Multiplicative (`*`, `/`, `%`)
3. Additive (`+`, `-`)
4. Relational (`<`, `>`, `<=`, `≤`, `>=`, `≥`)
5. Equality (`==`, `!=`, `<>`, `≠`)
6. Logical AND (`&&`)
7. Logical XOR (`^`)
8. Logical OR (`||`)
9. Assignment (`<-`, `←`)

If and If else Statements	

`if ( condition )`
`  <indented statement(s)>`
`end if`
	The indented statement or group of statements are executed if the Boolean condition evaluates to `true`, otherwise are skipped over if the condition evaluates to `false`.

`if ( condition )`
`  <indented statement(s)>`
`else`
`  <indented statement(s)>`
`end if`
	The indented statement or group of statements following the `if` are executed if the Boolean condition evaluates to `true`, otherwise the statements following the `else` are executed.

Loop statements	

`for(start; condition; step)`
`  <indented statement(s)>`
`end for`
	The indented statement or group of statements is executed each time the condition is true. The order of execution is rigid - start only once, check the condition, if true, do the action, then step, check, action, step, check, action, step, stop when check is false.

`while (condition)`
`  <indented statement(s)>`
`end while`
	The indented statement or group of statements is executed each time the `while` condition is true. The action and step are both included in the indented statements, and can be listed in any order.

`do{`
`  <indented statement(s)>`
`} while (condition)`
	The indented statement or group of statements is executed immediately after the `do`, always executing **at least once**. The action and step are both included in the indented statements, and can be listed in any order. The statement(s) will be executed again each time the ending `while` condition is true.

Method Blocks	

`void grading (char grade)`
`  <indented statement(s)>`
`end grading`
	This is a void method (procedure) named **`grading`** which accepts a character parameter **`grade`**, with no return value

`procedure coffeeType`
`  (string coffee )`
`  <indented statement(s)>`
`end coffeeType`
	This is a procedure (void method) named **`coffeeType`** which accepts a string parameter **`coffee`**, with no return value

`int f (int x, int y)`
`  <indented statement(s)>`
`end f`
	This is a return method (function) **`f`** which accepts two integer variables x and y and returns an integer

One and Two Dimensional Arrays	

`int [] list`
 	
`int array list`
see practice test, question 24 for this notation style
	declares an array of integers - two different notation styles are demonstrated in the practice test

`int [] list <- {1,2,-3,4}`
`int [][] grid <-`
`  {{-1,2},{3,-4,5}}`
`string [] words <-`
`  {"Tom","Dick","Harry"}`
 	
`float [] nums <-`
`  {1.2,-2.9,0.53}`
	Arrays declared and instantiated with values

`print list[0]` result **1** `print list[2]` result **3** `print grid[0][0]` result **-1** `print grid[1][2]` result **5** `grid[1][2] <- grid[0][1] + 10` `print grid[1][2]` result **12** `print words[2]` result **Harry** `print nums[1]` result **-2.9**	Individual array elements accessed

`int[] list <- new int[3]`
`list[0] <- 15`
`list[1] <- -1`
`list[2] <- 99`
`print list[2]` result **99**
`list[2] <- -10`
`print list[2]` result **-10**
	This code creates a new integer array with slots for three elements, and then assigns values to those slots. It thens outputs one of the elements, reassigns that slot, and outputs the new value.

`list.length` result **4** `len(list)` result **4** `grid.length` result **2** (2 rows are in the grid) `grid[1].length` result **3** (3 elements are in row 1 of the grid) `len(words)` result **3** `nums.length` result **3**	Here are two ways to return the number of elements in an array structure: Use the length command to report the number of elements contained. Call a method **`len`**, which returns the length of the array parameter

`for(int x<-0;x<list.length;x<-x+1)`
`  print list[x]`
 	
Output result: **1 2 -3 4**
 	
`for(int x<-0;x<len(words);x<-x+1)`
`  print words[x]`
	This for loop outputs all of the elements of the array.

Output result: **`Tom Dick Harry`**
 	
`for(int r<-0;r<grid.length;r<-r+1)`
`for(int c<-0;c<grid[r].length;c<-c+1)`
`print grid[r][c]`
 	
Output result: **`-1 2 3 -4 12`**	

`int[] list <- new int[3]` `list[2] <- 4`	Assigns the value 4 to the last position of the array
Note: 1D and 2D arrays for all data types will follow the same structure and notation as the given examples have shown.	

PARAMETERS

Parameter - A value, expression, or variable located inside the parentheses of a method call or method header.

> **Actual Parameter –** Value, expression, or variable in a method call.
>
> **Formal Parameter –** Variable (no literal values or expressions) listed in a method header that receives a value or expression result from an actual parameter in a method or procedure call, and is only in scope during the execution of the method.

Parameter Passing - The process in the source code of a program whereby data is passed from one section of the program to another, using method calls that send **actual parameters**, and method definitions that receive the passed data using **formal parameters**.

> **Passing by Value -** When a parameter is passed by value to a method, any changes made to the formal parameter inside the method **DO NOT** affect the actual parameter.
>
> **Passing by Reference -** When a parameter is passed by reference, the **memory location** originally referred to by the actual parameter is passed to the formal parameter, such that any changes made to the object by the formal parameter **also affect** the original actual parameter.

Searches

- Linear Search
  - A linear search uses a loop process to search for the target item, starting at the front of the list and proceeding through the list, one item at a time, stopping when it either finds the target, or reaches the end of the list.
  - There are three scenarios for the running time of a Linear Search:
    - Best Case - Constant time – O(1) – the search target is the first element of the list, or pretty close to the front
    - Average Case – Linear time – O(N) – the item is somewhere in the middle of the list. *Even finding the element halfway through the list is still considered Linear time.*
    - Worst Case – Linear time – O(N) – the item is at the end of the list, is pretty close to the end of the list, or is not found.
  - Pseudocode for Linear Search

```
int linearSearch(int[] list, int target)
    for (int j ← 0; j < list.length; j ← j + 1)
        if (list[j] == target)
            return j
        end if
    end for
    return -1
end linearSearch
//sample client code
int[] list ← {5,7,3,9,4,6}
print linearSearch(list,3)
print linearSearch(list,6)
print linearSearch(list,2)
```

```
2
5
-1
```

- Binary Search
  - A binary search requires the list to be in sorted order, according to the natural order of the data contained (alpha, numeric, etc.)
  - It then systematically
    - looks at the middle of the list to see if that data item matches the target.
    - If the middle item matches the target, return the position in the list where it was found.
    - If it does not match, determine which way to continue the search, either to the left of the middle if the target is less than the middle, or to the right if it is greater.
    - Within the left or right half of the list, it repeats the process:
      - Find the middle of that half
      - Compare it to the target
      - If it matches, return the position of that item, and if not, look left or right
  - This "divide and conquer" approach continues until the item is found, or until there are no more data items to examine, at which time the search process ends.
  - The three scenarios for the running time of a Binary Search are:
    - Best Case - Constant time – O(1) – the search target is the first "middle" element of the list
    - Average Case – Logarithmic time – O(log N) – the item is somewhere in the list
    - Worst Case – Logarithmic time – O(log N) – the item is not in the list.

  - Pseudocode for Binary Search

```
int binarySearch(int[] list, int target)
  int left ← 0,right ← list.length–1   //left at start, right at end
    while (left <= right)              //while left and right have not crossed
      int middle ← (left + right)/2    //find the middle of the list
      if (target == list[middle])
           return middle
      else
      if (target < list[middle]) //if target less than middle
         right ← middle – 1          //look left (change right marker to
                                      //position one place before the middle)
      else
         left ← middle + 1           //look right (change left marker to
      end if else chain
    end while
  return -1 //target not found
end binarySearch

//sample client code
int[] list ← {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20}
print binarySearch(list,3)
   //found in 5 steps – middle values of 10, 4, 1, 2 and finally 3
print binarySearch(list,16)
   //found in 4 steps – middle values of 10, 15, 18, 16
print binarySearch(list,21)
   //middle values of 10, 15, 18, 19, 20 are checked before left and right finally cross,
     indicating the value is not found.
```

```
3
16
-1
```

Sorts

- Selection Sort - Uses a nested loop process (loop within a loop) to "select" the best data for a particular position, then swap the best data with the value in that place.

```
void selectionSort(pass-by-reference int[] list)
  for (int k ← 0;k<list.length-1;k←k+1) //k loop starts at zero and stops
                                        //one position short of end of list
    int best ← k          //assume that position k has best value
    for (int q ← k+1;q<list.length;q ← q+1)//q loop searches the rest of
                                           //the list for a better value
          if (list[q] < list[best])    //if q indicates a better value,
            best ← q                   //best gets q
      end q loop
    swap(list, k, best)    //swap values at positions k and best
  end k loop
end selectionSort
//three step swap process
void swap (pass-by-reference int[] list, int i, int j)
  int temp ← list[i]
  list[i] ← list [j]
  list [j] ← temp
end swap
```

- Bubble Sort - Uses a nested loop process (loop within a loop) to compare neighboring items and make swaps as neighbors are out of natural order.

```
void bubbleSort(pass-by-reference int[] list)
  boolean done ← false //assume sorting process is NOT done
  while(not done)
    done ← true    //assume sorting process IS done, and list is sorted...BUT...
    for (int k←0; k<list.length-1; k←k+1) //...look through the list anyway to see
                                          //if any neighboring elements are out
                                          //of order and need to be swapped
      if (list[k+1] < list[k]) //if value of next-door neighbor is
                               //less than current value(neighbors are "out of sorts")
         swap(list, k, k+1)     //swap the two values
         done ← false           //set done to false
    end for k loop
  end while loop
end bubbleSort
//three step swap process
void swap (pass-by-reference int[] list, int i, int j)
  int temp ← list[i]
  list[i] ← list [j]
  list [j] ← temp
end swap
```

- Insertion Sort - Uses a nested loop process (loop within a loop) to “insert” each unsorted item in a list into its proper position.

```
void insertionSort(pass-by-reference int[] list)
  for (int k←1; k < list.length; k++)//outside loop starts at ONE and stops
                                          //at end of list
    int temp ← list[k]   //save copy of current value to be inserted
    int best ← k         //assume best place is at current position...BUT...
    while (best > 0 AND temp < list[best - 1]) //...keep looking for a better place
      list[best] ← list[best - 1]  //shift values over one spot while the previous
      best ← best-1                //place is better and haven't reached the start
                                     //of the list
    end while
    list[best] ← temp //insert the current value in the best place
 end for
end insertionSort
```

  - Compare this version to example on page 28 of TExES 141 Preparation Manual, CS 8-12

- The Merge Sort - Uses a “divide and conquer” recursive approach, repeatedly dividing the list in half until it is looking at only two individual items, merges the two elements back into one sorted list, working back up through each division, merging each pair of sorted sections together into the larger section, until the entire list is has been merged back into sorted array.

```
void mergeSort(pass-by-reference int[] list) //method for initial sort call
  int n ← list.length
  int[] temp ← new int[n] //creates a temporary array the same size as list
  mergeSortHelper(list, 0, n - 1, temp)
end mergeSort

//method that does the “divide and conquer” recursive process, receiving two arrays and two integers
//indicating front and back of portion of the current portion of the list being sorted
void mergeSortHelper(pass-by-reference int[] list, int front,
                     int back, pass-by-reference int[] temp)
  if (front < back) //if front and back positions have not crossed and are not the same
    int mid ← (front + back)/2 //find the middle position between front and back
    mergeSortHelper(list, front, mid, temp)//sort the left side, from front to mid
    mergeSortHelper(list, mid + 1, back, temp)//sort the right side, mid+1 to back
    merge(list, front, mid, back, temp) //merge the left and right lists
  end if
end mergeSortHelper

//method that merges two previously sorted lists, receiving two arrays and three integers
//indicating the front, mid, and back of the current portion of the list being sorted.
void merge(pass-by-reference int_array list, int front, int mid,
           int back, pass-by-reference int_array temp)
  int i ← front     //i marks the front of the left side of the list being sorted
  int j ← mid + 1   //j marks the front of the right side of the list being sorted
  int k ← front     //k marks the front of the temporary list, into which the values
                    //will be merged
  while (i <= mid && j <= back) //while neither front marker, i or j, have reached
                                //the end of their half of the list
    if (list[i] < list[j])  //if i’s value is less than j’s value
      temp[k] ← list[i]     //put i’s value into the temporary list, and
      i ← i+1                 //step i one position to the right
    end if
    else
      temp[k] ← list[j];    //otherwise, put j’s value into the temporary list, and
      j ← j+1                 //step j one position to the right
    end else
    k ← k+1                   //step k one position to the right
  end while
  //at this point, one of the two halves of the list has been completely loaded into the
  //temporary array, and the remaining half needs to be “cleaned out”

  //if i has not reached the middle, then the left half needs to be “cleaned out” and loaded
  //into the temporary list
  while (i <= mid)
    temp[k] ← list[i] //put i’s value into the temporary list
    k ← k+1       //step both k and i
    I ← i+1      //one place to the right
  end while
  //if j has not reached the back, then the right half needs to be “cleaned out” and loaded
  //into the temporary list
  while (j <= back)
    temp[k] ← list[j] //put j’s value into the temporary list
    j ← j+1    //step both k and j
    k ← k+1    //one place to the right
  end while
//load all temporary values in this current list back into the original list
  for(int x ← front; x <= back; x ← x+1)
    list[x] ← temp[x]
end merge
```

- The Quick Sort also uses a “divide and conquer” recursive approach, using a pivot value to create partitions, and then moves all values to the appropriate side of the pivot, repeating the process with each partition until the dividing process reaches a single element, at which time the recursive process ends and returns to the previous level, until the list is sorted.

```
void quickSort (pass-by-reference int[] list, int lo, int hi)
  if (lo >= hi) //if the current values of lo and hi have met or crossed,
    return          //do not continue this process
  end if
  int left ← lo  //left marker gets low end of current portion of list being sorted
  int right ← hi //right marker gets high end of current portion of list being sorted
  int pivot ← list[(lo+hi)/2] //pivot gets middle position value of current list

  while ( left < right)  //while left and right markers have not met or crossed
    while (list[left] < pivot) //look for a value to the left of the pivot that is on the
                               //wrong side, and stop there, with left marking that position
      left ← left+1
    end while
    while (list[right] > pivot) //look for a value to the right of the pivot that is on the
                                //wrong side, and stop there, with right marking that position
      right ← right-1
    end while
    if (left <= right)   //if the left and right positions have not crossed, swap the two
                         //values at those positions, and…
      swap (list, left, right)
      left ← left+1  //…step left marker one position to the right, and…
      right ← right-1//…step right marker one position to the left
    end if

  end while
  quickSort (list, lo, right) //quicksort the left side of the list
  quickSort (list, left, hi) //quicksort the right side of the list
end quickSort

void swap (pass-by-reference int[] list, int i, int j)
  int temp ← list[i]
  list[i] ← list [j]
  list [j] ← temp
end swap
```