CPS 1000 Assignment

This is an individual assignment and carries 100% of the final CPS 1000 grade.

Important instructions (read carefully and thoroughly):

The firm submission deadline is Friday 5th February 2016. Both a hard-copy of the report and the accompanying digital media should be handed in at Ms Vanessa Borg's office - FICT building, Level 1, Block B, Room 3. Make sure to consult the student hours in order not to miss the deadline if submitting on the final day.

A backup submission, that includes a soft-copy of the report and all files submitted on digital media, should be uploaded to the VLE by the same deadline. All files must be archived into a single .zip or .tar.gzip file.

You are to allocate *80 hours* for this assignment. Students that do no carry out lab exercises should expect the actual time taken to complete this assignment to be considerably longer.

This assignment consists of multiple tasks. Individual tasks will be introduced in class whenever sufficient content to carry them out has been duly covered. The first task will be introduced early November.

Only students that regularly attend lab sessions are entitled to assistance from lab tutors. Tutors are strictly not allowed to spell out solutions, rather they will be pointing you towards theory/lab exercises to assist you in carrying out assignment tasks. You will also be given continuous hints about how best to carry out the assignment tasks during lectures.

The first page in your assignment report must be the title of your assignment clearly showing your name, surname and study unit code. The second page must the table of contents. The report organization should follow the task sequence described in the following pages.

The majority of the marks will be assigned for proper code explanations rather than for functional binaries. Therefore all source code must be properly presented in the report along with the necessary code fragment explanations. The full source code must also be supplied on an accompanying digital medium along with the *makefiles* for each task. A `readme.txt` file must be included in the root folder describing the content's organization.

Reports that are difficult to follow due to low quality in the writing-style/organization/presentation will be penalized.

<u>Task 1:</u> <u>Problem solving</u> (Total: 50 marks)

Warming up

1a. Write a program that requests the hours worked in a week and then prints the gross pay, the taxes, and the net pay. Assume the following:

i. Basic pay rate = Eur 22.50/hr ;
ii. Overtime (in excess of 40 hours) is paid at x1.5 the basic pay;
iii. Tax rate: 0% of the first Eur 300, 15% of the next Eur 180, 29% of the rest.

Given frequent tax law changes the source code must be written with ease of maintenance in mind.

1b. Write a program that reads input as a stream of characters until encountering EOF. Have it report the average number of letters per word as well as the ratio of upper:lowercase letters for the entire stream. You don't count whitespace or punctuation as being letters in a word.

1c. Write a function `to_base_n(x,n)` that transforms `x` to its base-n equivalent; where `x` is a decimal number and `n` is a decimal expressible as 2^y.

1d. Write a function `str_reverse()` that reverses any string passed to it as argument.

(4 marks each)

Sort algorithms

1e. Write a function, `naive_sort()` that takes as a parameter an array of randomly generated integers and returns them sorted. Research and use the "insertion sort" method, an intuitive sorting algorithm which however scales badly with the size of the input array.

1f. An improvement on the previous sort function is to use "quick sort" instead, which takes a divide-and-conquer approach to the sorting problem by splitting up the array into smaller re-organized chunks and passes them individually to the *same* routine. Research this sort algorithm and use it to implement the `smarter_sort()` function.

1g. Write a single test application for both the above functions, taking care of generating the random array content as well as verifying their correct operation.

1h. Enhance the previous test application with a performance profiling routine that generates varying input array sizes of 10, 100, 1000, 5000, and 10000, timing the running times for both functions in each case. Plot a single graph comparing the running times for both functions.

(4 marks each)

Classic puzzles re-visited

1i. Passengers making use of the Maltese public transport service (*tal-linja*) look forward to kill journey times by counting stray cats observed through the bus windows. Because of this, *tal-linja* takes a keen interest in the health of the local stray cat population and decides to take a tally of the number of cats observed between each pair of stops as reported by passengers to their employees stationed at each stop. Unfortunately, the job of computing the grand total per journey is complicated by a primitive communication system where each bus stop employee can call only its immediate neighboring bus stop as opposed to all employees contacting the employee at the final stop stationed at the bus terminus. The implication is that the running total must be communicated all along the way. Write a program that simulates such a communication sequence *as initiated at the final bus stop, using recursion*. Passenger reports can be either be prefixed inside an array or simply ask the user to simulate the reported counts.

(8 marks)

1j. As the programmer of a vending machine controller your are required to compute the minimum number of coins that make up the required change to give back to customers. An efficient solution to this problem takes a d*ynamic programming* approach, starting off computing the number of coins required for a 1 cent change, then for 2 cents, then for 3 cents, until reaching the required change and each time making use of the prior computed number of coins. Write a program containing the function `ComputeChange()`, that takes a list of valid coins and the required change. This program should repeatedly ask for the required change from the console and call `ComputeChange()` accordingly. It should also make use of "caching", where any previously computed intermediate values are retained for subsequent look-up.

(10 marks)

Task 2: Data structures (Total: 20 marks)

The stack data structure (see Figure 1) is a container data structure that stores a collection of data items where the last added item is the first to be removed (LIFO – last in, first out). Adding items to the stack is referred to as a *push* operation, whilst removing the last-added item is referred to as a *pop* operation.

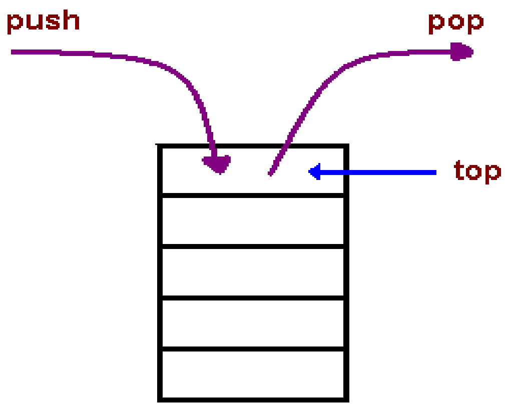

Figure 1: The stack data structure

A stack machine is a real or emulated computing machine characterized by how it evaluates expressions. Focusing on arithmetic expressions, a stack machine would take the "infix" expression `(4+(5*6))` and first converts it to its "postfix" equivalent `4 5 6 * +`, precisely denoting that the multiplication sub-expression is to be evaluated before addition. Similarly `((4+5)*6)` would be converted to `4 5 + 6 *`, where this time it is the addition sub-expression that will be evaluated first. Once in postfix notation a stack machine evaluates expressions by pushing operands on the stack, scanned left-right, and evaluating each sub-expression as soon as an operator is encountered. Sub-expression evaluation proceeds by popping the required number of operands off the stack and pushing back the result, as per example shown in Figure 2.

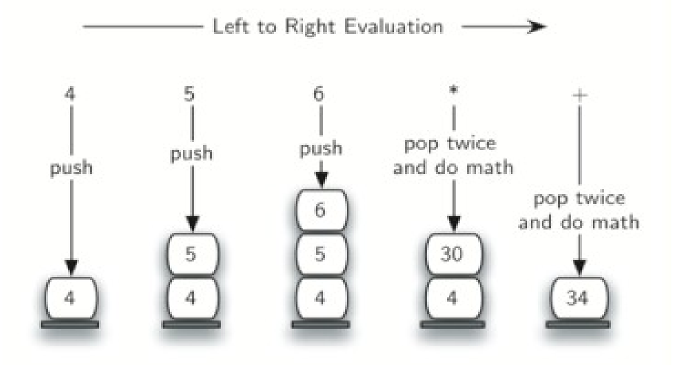

Figure 2: Stack machine computation

2a. Make use of a stack data structure implemented as an array in order to validate input infix expressions in terms of balanced parenthesis e.g. `( ( ) )` and not `) ( ( )`. Balanced parenthesis can be validated by:

- Scanning input express left-right,
- Pushing all left parenthesis on the stack, and
- Matching each right parenthesis with the top of the stack, popping the latter accordingly in the case of a left-right parenthesis match.
- Balanced expressions terminate with an empty stack, whilst unbalanced ones either leave extra parenthesis on the stack or encounter an empty stack when parsing a right parenthesis.

(5 marks)

2b. Make use of a stack data structure implemented as an array in order to convert infix expressions to postfix. Only consider arithmetic operations involving `+ - * /` (all binary). The conversion essentially involves shifting operators out of their immediate enclosing parenthesis. In the case of `((4+5)*6)`, the + is shifted from `(4+5)` to `(4 5)+` whilst the `*` is shifted to after the outermost parenthesis pair, resulting in the final `((4 5)+ 6)*`. Given that postfix notation does not require the use of parenthesis, the final result is actually `4 5 + 6 *`. Notice how the operands sequence remains the same, with just a shift in the operator positions occurring. There could be cases where operators are also re-ordered as in the case of `(4+(5*6)) → 4 5 6 * +`.

This conversion process is best implemented using a stack data structure for operators: the `opstack`.

- Expressions are scanned left-right.
- Operands are simply appended to the resulting postfix expression.
- On the other hand parenthesis and operators follow the procedure:
  - Push left parenthesis on the `opstack`.
  - On encountering a right parenthesis pop the `opstack` until the corresponding left parenthesis is removed. Append each popped operator to the end of the output list.
  - On encountering an operator push it on the `opstack`, however first pop and append to the output any operator with equal or higher precedence, or until a left parenthesis is encountered.
  - When expression scanning has terminated, pop and append to the output expression any remaining operators.

(10 marks)

2c. Write a main function that accepts input infix expressions from the console and proceeds by validating and converting arithmetic expressions to postfix format. Finally the postfix expressions are evaluated as described above and the results sent to the console.

(5 marks)

Task 3: Allocated memory and I/O (Total: 5 marks)

Make the previous program more memory efficient and practical to use by making use of dynamic arrays and making the program accept a sequence of expressions for evaluation from an input file. Likewise, evaluation results are to be written to an output file.

In your report only focus on the modified code.

Task 4: Abstract data types (ADT) (Total: 15 marks)

4a. Make your code more reusable and maintainable by reimplementing the stack data structure as an ADT that exposes the following interface:

`stack(max_size)`: creates a new stack with at most `max_size` elements.
`push(S, p)`: pushes element `p` on stack `S`.
`pop(S)`: pops the top element from stack `S`.
`top(S)`: returns, but without removing, the top element from stack `S`.
`is_empty(S)` : returns `true` if stack `S` contains no elements; `false` otherwise.

(5 marks)

4b. At the same time further improve memory efficiency by re-implementing the stack as a linked list.

(5 marks)

4c. Update the previous program accordingly to use the ADT implementation of the stack.
(5 marks)

Task 5: Shared libraries (Total: 10 marks)

Make your ADT even more re-usable and maintainable by shifting its implementation to a shared (dynamically linked) library. In the report focus on documenting the process of creating and linking to the shared library.

(10 marks)

END.