

## **Assessment of Programming Skills of First Year CS Students: Problem Set**

Notes to the working group participants. Enclosed in this file are the three problems. They are in ascending order of difficulty. As a baseline, we think our second semester students should be able to do problem 3 in 1.5 hours. To maintain consistency, the problems (no matter which you choose) should be given in the following manner.

- This is individual work.
- The work is to be done in a closed lab (meaning proctored with the students doing the work in the allotted time).
- The student's job is to produce a working, tested, etc., program in the time allotted.
- This is a programming exercise, the expectations are that the students will produce a program. Any design documentation, though important to solving the problem, is not important to this assessment.
- What follows can be cut and pasted as the assignment. Note the introduction is applicable to all three problems.

## Introduction ( 10 minutes to read and understand )

There are two main notations for entering information into hand-held calculators; Hewlett Packard calculators for the most part implement a Reverse Polish Notation (RPN), or “post-fix” notation, whereas Texas Instruments calculators implement the traditional “infix” notation. The major difference in the two styles is the order in which the mathematical operations are entered and executed.

RPN is a stack-based method of processing input. Numbers are read in and processed in the reverse order in which they are entered, while the operations themselves are read in and processed sequentially. “Infix” notation is the traditional method that most modern textbooks have adopted. It is mainly a symbol-based method of processing mathematical expressions, making use of a set of precedence rules defining the order in which values are processed by encapsulating operators and symbols.

RPN has a clear advantage over “infix” notation because RPN offers a simple implementation, naturally avoids ambiguity, and does not require parentheses. However, unless the user is well practiced in the art of RPN, it is not automatically intuitive in its use. For example, the “infix” notation for a particular formula is:

$$-(4 + (6 * (-2)) - 15) \quad (1)$$

However, in RPN, the same expression yields the following formula:

$$4\ 6\ 2\ \sim\ *\ +\ 15\ -\ \sim \quad (2)$$

In RPN, the numbers on the bottom of the stack are processed first, and then “work their way up” as the values are reduced toward the top. Clearly, the more intuitive format from an academic standpoint is “infix” notation (Equ. 1). However, you must deal with issues of precedence; in general, this is not as easy to program as the RPN-style of input (Equ. 2).

Other examples of “infix” and RPN style expression pairs are as follows:

$$4 * -(4 + 10 / 2.5 - 8) \quad (3)$$

$$4\ 4\ 10\ 2.5\ /\ +\ 8\ -\ \sim\ * \quad (4)$$

$$17.2 + 21 * 3 / (-7) + 21 \quad (5)$$

$$17.2\ 21\ 3\ *\ 7\ \sim\ /\ +\ 21\ + \quad (6)$$

$$((-(2 + 3 * 8))^4) * 7 \quad (7)$$

$$2\ 3\ 8\ *\ +\ \sim\ 4\ \wedge\ 7\ * \quad (8)$$

## Problem Set #1: Programming an RPN Calculator

DIFFICULTY LEVEL: 2

### Problem Statement:

You must write a program that reads, parses, and solves RPN-styled equations. You may assume that input will come directly from a terminal's standard input (keyboard or related device) and that the output should be directed to standard output for that terminal (monitor or related device). You may implement this program using only standard library routines provided by the language that is being used; no proprietary or other such libraries are allowed.

Your RPN expressions should be able to read the following operators. These operators' descriptions are provided for clarity.

- + Addition Operator. Adds two operands together.
- Subtraction Operator. Subtracts two operands together.
- \* Multiplication Operator. Multiplies two operands.
- / Division Operator. Divides two operands, in the order they are placed in the stack.
- ~ Inverse Operator (Negative Operator). Takes the inverse of the current value in the buffer.
- ^ Power Operator. Multiplies value in buffer by itself by a number of times specified by the operand that follows.

Each line of input will be entered in the relative form of:

{ numbers } { operators }

Each number and operator may be assumed to be separated by some form of whitespace to make parsing the input easier. This input should be entered on an input line non-interactively; In other words, the program can NOT query the user for any EXTRA information pertaining to the contents or makeup of the expression. The only interactive element the program may use in the expression input process is a prompt to indicate the program is ready for input. A sample session is below; the program should terminate when an input contains only the letter 'q'. If there is an error with the input, the program should state such and begin accepting the next expression. At the end of each calculation, the calculator is considered to be cleared, and the stack should be emptied. Floating point arithmetic should be assumed and programs should allow non-integer expressions as valid input.

```
{unix: user: !} ./runprog
> 2 2.00 3 * +
ans = 8
> 4.00 5 2 ^ - ~ +
ERROR: Expression Invalid.
> 4 21 3 / - ~
ans = 3
> q
```

## **Problem Set #2: Programming an “infix” Calculator w/o precedence**

---

**DIFFICULTY LEVEL: 1**

### **Problem Statement:**

You must write a program that reads, parses, and solves “infix”-styled equations. You may assume that input will come directly from a terminal’s standard input (keyboard or related device) and that the output should be directed to standard output for that terminal (monitor or related device). You may implement this program using only standard library routines provided by the language that is being used; no proprietary or other such libraries are allowed.

Your “infix” expressions should be able to read the following operators. These operators’ descriptions are provided for clarity.

- + Addition Operator. Adds two operands together.
- Subtraction Operator. Subtracts two operands together.
- \* Multiplication Operator. Multiplies two operands.
- / Division Operator. Divides two operands, in the order they appear.
- ^ Power Operator. Multiplies value in by itself by a number of times specified by the operand that follows.

Each line of input will be entered in the relative form of:

NUM OP NUM OP NUM ...

Where NUM and OP represent Numbers and Operators, respectively.

Each number and operator may be assumed to be separated by some form of whitespace to make parsing the input easier. This input should be entered on an input line non-interactively; In other words, the program may NOT query the user for any EXTRA information pertaining to the contents or makeup of the expression. The only interactive element the program may use in the expression input process is a prompt to indicate the program is ready for input. A sample session is below; the program should terminate when an input contains only the letter ‘q’. If there is an error with the input, the program should state such and begin accepting the next expression. At the end of each calculation, the calculator is considered to be cleared, and the memory (if any) should be emptied.

This program does NOT have to consider precedence, nor contain any parenthesis. This will lead to some mathematically incorrect answers, but to keep things simple we will not be enforcing precedence. Floating point arithmetic should be assumed and programs should allow non-integer expressions as valid input. A sample session is below:

```
{unix: user: !} ./prog
> 2 + 3 * 4 - 5
ans = 15
> 2 + 3 ^ 2 * 4 * -1
ans = -100
>q
```

## Problem Set #3: “Infix” Calculator with simple precedence

---

DIFFICULTY LEVEL: 3

### Problem Statement:

You must write a program that reads, parses, and solves “infix”-styled equations. You may assume that input will come directly from a terminal’s standard input (keyboard or related device) and that the output should be directed to standard output for that terminal (monitor or related device). You may implement this program using only standard library routines provided by the language that is being used; no proprietary or other such libraries are allowed.

Your “infix” expressions should be able to read the following operators. These operators’ descriptions are provided for clarity:

- ( )     Parenthesis. Used to group numbers and operators to give a simple order of precedence. Expressions contained within parentheses must be evaluated before expressions outside of parenthesis.
- ^     Power Operator. Multiplies value in by itself by a number of times specified by the operand that follows.
- \*     Multiplication Operator. Multiplies two operands.
- /     Division Operator. Divides two operands, in the order they appear.
- +     Addition Operator. Adds two operands together.
- Subtraction Operator. Subtracts two operands together.

Each line of input will be entered in the relative form of the following examples:

```
> ( 2.0 + 3 ) * ( 5 * 2.0 * -1 )
ans = -50
> ( 3 + 1.0 * 2 ) ^ ( 2 + 1 )
ans = 512
> ( ( 3 + 1 ) * 2 ) ^ ( 9.0 / 3 )
ans = 512
```

Each number, operator, and parentheses may be assumed to be separated by some form of whitespace to make parsing the input easier. This input should be entered on an input line non-interactively; In other words, the program can NOT query the user for any EXTRA information pertaining to the contents or makeup of the expression. The only interactive element the program may use in the expression input process is a prompt to indicate the program is ready for input. A sample session is below; the program should terminate when an input contains only the letter ‘q’. If there is an error with the input, the program should state such and begin accepting the next expression. Floating point arithmetic should be assumed and programs should allow non-integer expressions as valid input.

```
{unix: user: !} ./prog
> ( 2 + 3 ) * ( 4 - 5 )
ans = -5
> ( ( 2 + 3 ) ^ 2 * ( 4 * -1 )
ans = -100
>q
```

At the end of each calculation, the calculator is considered to be cleared.

This program does NOT have to consider operator precedence, but must consider parenthesis. This may still lead to some mathematically incorrect answers, but to keep things simple we will only be enforcing parenthesis precedence.