CSE213L: Data Structures and Algorithms
LNMIIT, Jaipur
Training Set 01

Overview and Organisation of DSA Labs

DSA Labs are organised as a collection of 10 training sets. These training sets are self-paced, and students must avoid temptation to rush to a later (advanced) training set before fully learning the lessons and skills from the earlier training sets. Submitting a solution prepared by another student may deceive a tutor, but such progress will hurt you at the examinations and at the other occasions.

Each training set is made of three tasks. The tasks progressively build on the previous tasks. For the best learning outcomes, you will be required to demonstrate each completed task during a lab session and get your completion recorded and signed (with date and time) in your lab record book by a tutor. Each student is required to maintain a lab notebook to record their progress in their DSA Labs.

For online offering of DSA lab, students will be assigned a tutor who will guide the students and validate record their progress. In this mode the students will self-record their progress. However, students will be randomly selected (several times during the semester) to justify their claims. If the student fails to convince the tutor of the correctness of their claim of completed work, the training set and all following training sets recorded as completed by the student will be assigned 0 marks. The instructor may schedule class quizzes for this purpose.

Even in the online mode of the course offering, lab examinations are expected to be organised in the institute laboratories. All lab examinations may be scheduled over a short period to keep the students' on-campus stay and visits short.

Normally, in any physical lab session on LNMIIT campus of 2 hours only 4 task completions will be recorded. This restriction ensures that each student progresses through the lab sessions in a steady and efficient fashion. Trying to do too many tasks in a single lab session is not the best approach to learning new skills.

Occasionally, some students may delay completion of a task and progress to a new training set. We may accept this as an infrequent act. However, such missed tasks should be completed in the following lab session.

Tasks in each training set are designed to be completed in the set sequence. Each new (later) task is helped by the work completed for the previous tasks of the training set.

Students will progress on the training sets based on their abilities and their other needs. The aim of the lab exercises is to learn the skills and it is not to rush through the topics. Not every student in a class needs to work on the same training set or task in a lab session. Personal learning needs and preferences of each individual student will determine how they manage their lab training in a rational and rewarding way.

Students who progress over the assigned training sets matching the expected schedule of topics in the related theory course (DSA) will be at an advantage in getting marks for the internal work. The first training set needs to be completed within 14 days from the start of the term to receive the full credit of marks for the training set. The later training sets have their

due dates for receiving the full credit set one week apart counting from the due date for the first training set. A clear calendar will be available when institute's academic calendar for the semester is available.

Student attendance is compulsory at all lessons for LNMIIT students. DSA-Lab instructors expect every student to be present at every scheduled lab session. This requirement has an essential equivalent for the online course too. Each physical attendance should be preceded by meaningful preparations for the lab work. In addition, the student should work on the lessons learned in a lab by necessary post lab reviews. The lab instructors will meet and discuss with each student a few times during the semester to monitor, review and record (to assign assessment marks, if appropriate) for their progress. This requirement is recorded in the course CIF.

To summarise:

1. Each lab training session is made of three tasks.
2. Lab tutors record, completion of each task for the purpose of course assessment.
3. Extra credit is available for completing a training set by its last scheduled date.
4. The extra credit dates give set very gentle pace of progress. The students who seek to complete all training sets before the final lab examination period must work much more rapidly than these dates.
5. No more than 4 task completions are recorded for any single lab session.
6. Physical attendance of the students will be recorded in each session. Every lab session is essential. Absence from the lab sessions is firmly disapproved.
7. Lab staff maintains academic records to ensure that the students are coming to the labs well prepared and the student is completing the lab work diligently and sincerely.
8. Students will progress through the lab training exercises at a pace that suits their learning needs and abilities.
9. Interviews with the instructors or quizzes will be used to advice the students if it will help the student to repeat some training lessons if they are found to be lacking in some skills.

End of course overview section.

Simple Infix Arithmetic Expressions with Integer Operands

An arithmetic expression made of integers is a remarkably simple structure. Each of these are arithmetic expression:

1. An integer, with a possible minus or plus sign in front, is an expression.
2. An expression enclosed in a pair of matching parentheses is an expression.
3. Two expressions joined by an arithmetic operator is an expression. The operators of interest, for this training set are add, subtract, multiply, divide and modulo.

And, *nothing else is an arithmetic expression!*

Let me emphasise here that minus and plus are unary operators. Add and subtract are binary operators. Unary operators are part of the operand (integer) value. Please do not process them as operators.

In this training set and later too, we require that DSA-L students use C `stdio` library functions to read the signed integer values. Reading digits and writing code to combine digits to compute the value of an integer is not a good programming culture.

You are advised to read about `stdio` functions `sprintf()` and `sscanf()` for this training set tasks. These functions support reading and printing of integer values through a string/array. More information about these functions is available on pages 245 and 246 of K&R ANSI C book. You may access the book at http://www2.cs.uregina.ca/~hilder/cs833/Other%20Reference%20Materials/The%20C%20Programming%20Language.pdf

In this training set we will develop a program to evaluate an arithmetic expression. This will be accomplished by first writing code to convert a given infix expression into an equivalent postfix form. And then evaluating this postfix representation of the expression. The evaluation and conversion algorithms are topics included in your DSA class. The algorithm steps are included at the end of this document for a convenient access.

Training set 01: Task 01

Listed below is a working version of program that reads an infix expression from the standard input stream and stores it in an array. The program then prints the token in the expression one by one. Most tokens have only their *kind* property associated with them. However, you will note that tokens of kind INT has an additional attribute `value` attached to them.

Terminology

Token

Token for an expression is a string of one or more characters that is meaningful single item. In the present problem these items are operands (integer values), parentheses, and operators.

A call to function `getToken()` is forwarded to the specialised functions depending on the expected `kind` of the next token in the expression. As a consequence, functions `getExpression()` and `getOperator()` take turns to read operand and operator tokens in this program.

Though we do not expect a bad expression as input, the introductory program code given below for this task, terminates reading of an expression if it runs into an unexpected situation.

Carefully study the program below. The program can read parenthesis-free expressions. In such expressions, operands and operators alternate.

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

#define LPAR '('
#define RPAR ')'
#define MINUS '-'
#define PLUS '+'
#define ADD '+'
#define SUBTRACT '-'
#define MULTIPLY '*'
```

```
#define DIVIDE '/'
#define MODULO '%'
#define FINISH '\0'
#define INT '0'

#define EXPRESSION 1
#define OPERATOR 0

struct token {
      char kind;
      int value;
};

char expr[100];
int where = 0;

void skipWhite() {
      while (isspace(expr[where])) where++;
}

void skipDigits() {
      if (expr[where] == '+' ||
            expr[where] == '-')
            where++;
      while (isdigit(expr[where])) where++;
}

struct token getExpr() {
      struct token token;
      skipWhite();
      if (expr[where] == PLUS || expr[where] == MINUS ||
                  expr[where] >= '0' && expr[where] <= '9') {
            token.kind = INT;
            sscanf(expr+where,"%d", &token.value);
            skipDigits();
      } else {
            printf("Bad symbol\n");
            token.kind = FINISH;
      }
      return token;
}

struct token getOperator() {
      struct token token;
      skipWhite();
      switch (expr[where]) {
            case ADD: case SUBTRACT: case MULTIPLY:
            case DIVIDE: case MODULO:
                  token.kind = expr[where];
                  where++;
                  break;
```

```
            case FINISH:
                  token.kind = FINISH;
                  where++;
                  break;
            default:
                  printf("Bad symbol\n");
                  token.kind = FINISH;
            }
            return token;
}


struct token getToken() {
      struct token token;
      // Initially expecting EXPRESSION
      static int expecting = EXPRESSION;
      int i;

      if (expecting == EXPRESSION) {
            token = getExpr();
            expecting = OPERATOR;
            return token;
      } else
      {
            token = getOperator();
            expecting = EXPRESSION;
            return token;
      }
}

int main (void) {
      struct token token;
      int i;

      printf("Input expression:");
      fgets(expr, 99, stdin);

      token = getToken();
      while (token.kind!=FINISH) {
            printf("%c", token.kind);
            // Print additional info for INT
            if (token.kind == INT)
                  printf("  %d", token.value);
            printf("\n");
            token = getToken();
      }
      return 0;
}
```

An expression with parentheses is not significantly more difficult to read. In this task, you will be taught to expand the given program code so that it systematically reads expressions with matching pairs of parentheses.

Though both specialised token reading functions (`getOperator()` and `getExpression()`) in the given program can return token FINISH, yet this token is normally expected only from function `getOperator()`. This idea can be easily used to correctly read two parentheses symbols. A left parenthesis can be read when we are expecting to read an operand. An operand is read by function `getExpression()`. And, the read left-parenthesis token LPAR could be returned and printed. After reading a LPAR token, the program continues to expect an expression in its input.

On the other hand, when the program is expecting to read an operator, the operator occurrence may be preceded by an occurrence of a right parenthesis. This parenthesis must match an open left parenthesis in a well-formed expression.

Also, in a well-formed expression, we can reach the end of an expression when we are expecting to read an operator.

It will require only about 10 lines of additional code in functions `getToken()`, `getExpr()` and `getOperator()` to be able to successfully read and produce tokens for arithmetic expressions that includes well-balanced parentheses pairs.

In your demonstration of completion of this task, print all tokens of the input expression with one token on each output line. My output for a program run is as follows (Note: the illustration expression has a negative number -34):

```
Input expression:100+29*(234-23)/(23-23--34)
0  100
+
0  29
*
(
0  234
-
0  23
)
/
(
0  23
-
0  23
-
0  -34
)
```

Training Set 01: Task 02

In Task 02 the trainee will develop a new program to evaluate a postfix expression. Many of the functions and utility codes developed in Task 01 is useful for this program too. Later in Task 03, the trainee will be asked to write a function to convert an infix expression into a postfix expression and combine solutions to Task 01 and Task 02 together to evaluate expressions.

1. Write a C program to evaluate a postfix expression that is entered as input through input stream `stdin`. See textbook for the algorithm, if needed, in your textbook: Thareja: Figure 7.23.
2. [Read tokens of Postfix expression] In completing this task, follow the coding patterns used in Task 01. Read the input postfix expression into an internal array and read operators and operands using `stdio` function `sscanf()`. You may assume that binary operators ADD (+) and SUBTRACT (-) will be separated, on both sides, from the other tokens of the postfix expression by a space. And, Symbol minus will always be immediately before the first digit of a negative INT value (there will never be a space between minus sign and the first digit of the value). Further hints for this dot point are provided below.
3. In completing program for this task, you must implement Stack ADT interface functions: `push(), pop(), peek(), isEmpty(), isFull(), initStack()`.
4. Test your program with several inputs.
    a. 10 20 + 30 *
    b. 20 10 – 5 /
    c. 25 -10 %

Hints for reading tokens of postfix expression

It is suggested that students add 2 at the end of each function name they write for this task. This will avoid confusion between the functions provided to the students in Task 01 and those written by the students as they complete Task 02.

We have declared token as structure with two parts: kind and value. Only operands (indicated by kind designator INT need value attribute. Tokens of all other kind require no additional attribute information.

Students can write code for function `getToken2()` in two stages. In the first stage, the students should develop this function to return only tokens with member kind. At this time, students can ignore the value attribute.

This is done by simply progressing the index `where` over array `postfixExpr[]` (the array is read from stream `stdin`). Index where is incremented over the white spaces. If `postfixExpr[where]` is '+', '*', '/', '%' or a digit it is easy to know the token kind. The harder part is if it is char '-'. We need to determine if the token is binary operator subtract or it is minus sign before a value. This can be done by looking at character `postfixExpr[where+1]`. Thus, `(postfixExpr[where] == '-' && postfixExpr[where+1] == ' ')` indicates token of kind `SUBTRACT`. And, `(postfixExpr[where] == '-' && postfixExpr[where+1] != ' ')` indicates start of an `INT` token.

Once, a student has successfully determined the token kinds in array `postfixExpr[]`, they can complete the code for function `getToken2()` by appropriately using `sscanf()` to extract attribute value for the tokens of kind `INT`.

Training Set 01: Task 03

1. Write and add a function to convert infix expression to a postfix expression to the program developed in Task 01. The algorithm is described in Reema Thareja's book in Section 7.7.3 page 232. This topic will also be discussed in DSA class.
2. First test your program by printing tokens on the output stream as per their order in the postfix expression.
3. Next, modify this program so that it prints these tokens separated by a single space. Print each operator as a single character symbol and each operand (integer) as a value using C function `printf()` with conversion code "`%d`".
4. Once you are satisfied by the correctness of your program, modify the program developed in step 3 above, so that it writes its output in a `char` array using `stdio` function `sprintf()`.
5. In the final development, combine your program with the program developed as Task 02 to evaluate the postfix expression derived from the infix expression read from the standard input stream. You have placed this expression in an array using `sprint()` in the previous step.

Vishu Malhotra
04 June 2020
The LNMIIT, Rupa ki Nangal
Jaipur, Rajasthan 302031

Appendix

Some snips from textbook by *Reema Thareja Data Structures Using C, Second Edition, Oxford University Press, New Delhi, 2014*

```
Step 1: Add a ")" at the end of the
        postfix expression
Step 2: Scan every character of the
        postfix expression and repeat
        Steps 3 and 4 until ")"is encountered
Step 3: IF an operand is encountered,
        push it on the stack
        IF an operator O is encountered, then
        a. Pop the top two elements from the
           stack as A and B as A and B
        b. Evaluate B O A, where A is the
           topmost element and B
           is the element below A.
        c. Push the result of evaluation
           on the stack
        [END OF IF]
Step 4: SET RESULT equal to the topmost element
        of the stack
Step 5: EXIT
```

Figure 7.23 Algorithm to evaluate a postfix expression

```
Step 1: Add ")" to the end of the infix expression
Step 2: Push "(" on to the stack
Step 3: Repeat until each character in the infix notation is scanned
       IF a "(" is encountered, push it on the stack
       IF an operand (whether a digit or a character) is encountered, add it to the
       postfix expression.
       IF a ")" is encountered, then
         a. Repeatedly pop from stack and add it to the postfix expression until a
            "(" is encountered.
         b. Discard the "(". That is, remove the "(" from stack and do not
            add it to the postfix expression
       IF an operator O is encountered, then
         a. Repeatedly pop from stack and add each operator (popped from the stack) to the
            postfix expression which has the same precedence or a higher precedence than O
         b. Push the operator O to the stack
       [END OF IF]
Step 4: Repeatedly pop from the stack and add it to the postfix expression until the stack is empty
Step 5: EXIT
```

Figure 7.22 Algorithm to convert an infix notation to postfix notation