EECS 211 Fall 2009
Description of Major Project for the Term

Introduction.

The remaining program assignments will build an application which translates simple C code into an intermediate form, called postfix form, and an evaluator for the postfix form. By the end of the project your program will be able to handle arithmetic and logic expressions as well as if, if-else, and while loops. The final assignment will be to translate and execute the following code:

```
VAR sume;  VAR sumo;  VAR j;  VAR a;   VAR b;
sume = 0;   sumo = 0;   j = 1;
while(j<11) {
    if( j/2*2 == j ) {
        sume = sume + j;  cout j;  cout sume;
    }
    else {
        sumo = sumo + j;  cout j;  cout sumo;
    }
    j = j+1;
}
a=5;
while( (b=1) (a=a-1) a>0 ) while(b<a) { cout a; cout b; b=b+1;}
```

This document describes the sequence of assignments and the classes we will develop. It also describes infix and postfix notations.

I have organized the sequence of assignments to illustrate several basic principles of software design and development. Of course, I have done almost all the design, but as each new assignment is given and also at the end of the project you should reflect on how each assignment adds a logically "next" piece to the overall project. Software development principles illustrated in this project include:

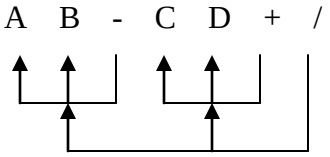
- top-down design and development;
- the utility of a good test harness at the beginning of the development;
- the benefits of modularization;
- the use of classes and objects to model a domain of interest.

Of course, the assignments give you practice in basic C++ programming techniques such as how to implement classes, how to use dynamic memory allocation, how to arrange data in basic data structures, how to process non-numeric data, and many more.

Background: Infix and Postfix Notations.

The notation we use for C++ expressions is called infix notation because for the standard binary operators (+, -, *, /, etc.) we write the operator in between the two operands. One consequence of writing the operator between is that we must introduce the notion of operator precedence and the use of parentheses to define the sequence of the operations in more complex expressions. We must also use additional punctuation symbols, such as semicolon and brackets, to delineate one statement from the next. There are two notations, called prefix and postfix, which require none of these extra symbols. That is, there are no parentheses or semicolons or brackets and no need for the notion of operator precedence. The only symbols in the expressions are the variables, constants, and operators. Moreover, prefix or postfix notations handle operators with arbitrary arity – unary operators, binary operators, ternary operators, etc. In infix notation, operators with three or more operands and functions require parentheses to enclose the argument list.

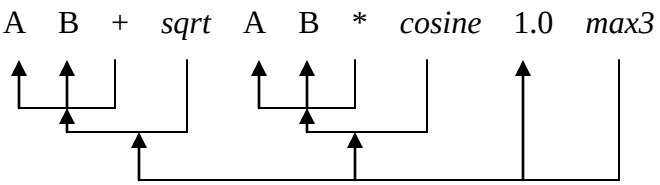
Our project will translate infix expressions into postfix notation. In postfix notation each operator has an arity, that is, the number of operands that it takes. The operator is written after the correct number of arguments, hence the name postfix. For example, the sequence A B + is the postfix expression for adding A and B. The first operand is the left-side operand, and the second operand is the right-side operand. Thus, A B - is the postfix for A minus B; similarly, A B / stands for A divided by B. Postfix expressions are evaluated (or "read") left to right. Thus, the expression is scanned left to right until an operator is found. Then that operator is applied to the previous values. Just as in infix notation, an operand may itself be an expression. Consider the following postfix expression:



The linkages show which sub-expressions go with each operator. The minus is applied to A and B, the plus to C and D, and the division to A-B and C+D. Note that this expression requires parentheses if written in infix notation: (A-B)/(C+D). The postfix expression:

A B C / - D +

corresponds to the infix expression A-B/C+D, i.e., without the parentheses. Finally, postfix notation handles operators of arbitrary arity. Suppose *max3* finds the maximum of three numbers. The postfix expression



represents *max3*(*sqrt*(A+B), *cosine*(A*B), 1.0). Note the need for parentheses in the infix notation.

Tentative Program Sequence.

Program 3: In this program we will develop two basic classes – symbol and symbol list. The symbol class will be used to hold variables, constants, operator symbols, etc. The symbol list class will be used to hold the postfix expressions and also to hold the list of variables declared in the postfix expressions.

Program 4: As you can see in the sample program in the Introduction, symbols may be more than one character. In program 4 you will write code to extract from an input file sequences of characters that form one token. For example, the input file might contain the sequence:

x15=42;

This sequence contains four tokens: x15, =, 42, and ;.

Program 5: In program 5 we will add three derived classes – variable, constant and operator – to hold the different information required for each of these kinds of symbols. We will also provide for the recognition and storage of the special symbols, such as the operator symbols, if, else, and while. Finally, we will include a function that converts a string representing an integer into the internal **int** format.

Program 6: In this program we will add a new class, stack of symbols, to hold the operator symbols during translation. Then we will write code that translates basic expressions (i.e., excluding if and while) from infix notation to postfix notation.

Program 7: Program 7 is a postfix expression evaluator.

Program 8: Finally, in the last program we will add the translation and evaluation of if and while statements, including the handling of { and }.