ESO207: Data Structures and Algorithms

Programming Assignment *1* *Due: 21 Aug 2014*

Problem 1. The input is an arithmetic expression in infix notation, for example, $24*(3+5)/7\%3$. The problem is to write a program that reads the expression given in a single input line, parses it, evaluates it and gives the output value as a single integer or prints the string "Input Error" if the input is malformed. The numbers in the expression are integers. The operators come from the set $\{+, -, *, /, \%, (,)\}$. In this version of the problem, we do not consider unary $-$ operator; the $-$ operator is the binary minus operator. The operators act on integers and return integers, as per their usual definitions. We assume the usual precedence among operators, that is, from lowest to highest, $\{+, -\}$, then, *, followed by $\{/, \%\}$ and parenthesis, where, + and binary $-$ are at the same level of precedence and have left to right associativity, so do $\{/, \%\}$. For this problem, assume that your input is well-formed, that is, there are no illegal input expressions. (75)

Problem 2.. Same as the previous problem, except that your work program should correctly identify and return an error if the input infix expression is malformed.

Problem 3. The problem is the same as the previous problem where now we additionally allow the unary $-$ operator. The $-$ operator appropriately signifies a unary minus operator or a binary minus operator depending on the context. We assume the usual precedence among operators, that is, from lowest to highest, $\{+, -\}, *, /$, unary $-$ and brackets. Unary operators associate right to left. Other operators, except parenthesis, associate left to right. Parenthesis have the highest precedence, and there is no need for an associativity rule for parenthesis. (25)

Discussion. In problems such as these, it might help to write a *tokenizer* module. The input may be thought of as divided into tokens, where each token is either a number or an operator. The goal of the tokenizer is to free the implementation of the actual calculator from parsing the input. Define a token structure that has a *TkType* field and a *TkVal* field. The *TkType* field takes one of two values, *Operator* or *Operand*– these values may be defined as global constants in the program. If the type of the token is *Operator*, then, the *TkVal* field can store the *opcode* (operator code) for that operator. For example, one may define the opcode of + as 1, opcode of $-$ as 2, opcode of unary $-$ as 5 and so on, as global constants in the program. If the type of the token is *Operand* then the *TkVal* field is the integer value of that operand. Note that it is the tokenizer that should correctly distinguish between the two operators, binary $-$ and unary $-$.

Once the tokenizer is written, one can focus on the problem at hand, which is to evaluate the expression. One approach could be to transform the expression into postfix, and then evaluate the postfix expression. The other approach is to directly evaluate the infix expression. If operators have opcodes, contiguously numbered from say 1 to k, then one can define the precedence relation by keeping an array *Priority* such that *Priority*[*opcode*] gives the priority of the operator whose code is *opcode*.

A standard way of evaluating infix expressions in a single pass over the input is to keep two stacks, one for the operands, and another for the operators. If the next token is an operand, then it is pushed onto the operand stack. If the next token is an operator, then, the decision is based on this operator. If it is '(' then it is pushed on stack. If it is ')', then, we pop an operator from the stack,

apply this operator to the corresponding operands on the top of the operand stack (depending on the arity of the operator) and push back the resulting value on operand stack. This is continued until the matching '(' is met. If the operator is something other than the brackets, then, we compare the current operator with the operator on the top of the operator stack. If the current operator is higher in precedence than the top of the stack operator (or equal in precedence), then the current operator is pushed onto stack. Otherwise (the current operator is lower in precedence than the current operator on the top of the stack), the operator on the top of the stack is evaluated using the operands on the top of the operand stack and the result is pushed back on to the operand stack. This process is repeated until either (a) the current operator is higher in precedence than the one on the top of the operator stack, or, (b) the operator stack has '(' on its top, or (c) the operator stack is empty. In all these cases, the current operator is pushed onto the operator stack. Finally, when the end of the input is reached then the operators in the stack are popped one at a time, and evaluated against the operands in the operand stack, the result is pushed back, until the operator stack is empty (also one should not meet a '(' on the stack. There should just be one value on the operand stack, which is the answer.