CSC 477/677 Analysis of Algorithms

Hints for Programming Assignment 1

Definition of the Problem

Write a computer program which takes as input an infix expression of length at most 80 characters, and outputs an equivalent postfix expression.

Infix Expressions

An infix expression consists of variables, operators, and parentheses.

- Each variable is written as one lower case letter.
- The only operator symbols are + and −.
- Addition is indicated by +.
- Subtraction is indicated by −.
- Multiplication is indicated by concatenation, just as in high school. For example, xy means x times y.
- Negation is indicated by −.
- The operator − indicates subtraction if possible, otherwise negation.
- Addition, subtraction, and multiplication are left-associative. Addition and subtraction have equal precedence. Multiplication has precedence over addition and subtraction. Negation has precedence over all other operations.

Postfix Expressions

A postfix expression consists of variables and operators.

- Each variable is written as one lower case letter.
- The operator symbols are +, −, ∗, and ∼, which indicate addition, subtraction, multiplication, and negation, respectively.

The Program

The program has four data structures.

- The *input file*, which is a string of symbols ending in an *end of line* symbol, which we call *eoln*. Each time *read* is executed, the leftmost symbol of the input file is deleted. Initially, the input file contains an infix expression, and when the program is finished, the input file is empty.
- The *stack*, which can contain only the symbols $+$, $-$, $*$, $\sim$, and (. Initially, the stack is empty, and when the program is finished, the stack is empty.
- The *output file*, a string of symbols. Each time *write* is executed, a symbol is appended to the output file. Initially, the output file is empty, and when the program is finished, the output file contains a postfix expression equivalent to the original input.
- The *state*. There are only two possible states, *expect expression* and *expect operator*. Initially, the state is *expect expression*, and when the program is finished, the state is *expect operator*.

Lookahead

The program can examine the leftmost symbol in the input file without reading it. In order to implement this feature, we need a *lookahead buffer* which consists of one symbol. If the input file is empty, the lookahead buffer contains a special *end symbol*, such as \$.

Peeking at the Top Item

The program can examine the top symbol in the stack without removing it. This feature can be implemented using the standard stack operators as follows:

```
topitem := Pop(S)
Push(topitem,S)
```

After executing that code, the variable *topitem* will be equal to the item at the top of the stack, but that item will still be on the stack.

Steps

The algorithm consists of *initialization*, followed by a sequence of *steps*. Initialization consists of setting the stack to be empty and setting the state to be *expect expression*. Write *next* for the next symbol in the input file. In the implementation of the program, the variable *next* will be the lookahead buffer.

The steps of the algorithm are defined as follows.

- If the state is *expect expression* and *next* is a variable, then read *next* and write it, and then change the state to *expect operator*.

- If the state is *expect expression* and *next* is (, then read *next* and push (onto the stack. The state does not change.
- If the state is *expect expression* and *next* is $-$, then read *next* and push $\sim$ onto the stack. The state does not change.
- If the state is *expect expression* and *next* is not one of the above choices, then the input was erroneous.
- If the state is *expect operator* and *next* is either $+$ or $-$, then pop off and write each operator in the stack down to the bottom or to the nearest (, whichever comes first. Then, read *next* and push it onto the stack, and then change the state to *expect expression*.
- If the state is *expect operator* and *next* is), then pop off and write each operator in the stack down to the bottom or to the nearest (, whichever comes first. If the stack is then empty, the input was erroneous. Otherwise, pop the stack, and discard both the left and right parentheses. The state does not change.
- If the state is *expect operator* and *next* is either (or a variable, then pop off and write each $\sim$ or $*$ in the stack down to the bottom or the nearest (, $+$, or $-$, whichever comes first. Push $*$ onto the stack, then change the state to *expect expression*. Do not read; the value of *next* does not change.
- If the state is *expect operator* and *next* is *eoln*, then pop off and write each operator in the stack down to the bottom or to the nearest (, whichever comes first. If the stack is then empty, the program is finished, and the output file is the correct postfix expression. If the stack is not empty, the input was erroneous.

Example Runs of My Program.

Infix Expression	**Postfix Expression**
x	x
$x + y$	$xy+$
$x - y$	$xy-$
xy	$xy*$
$(-x)y$	$x \sim y*$
$x - -y$	$xy \sim -$
$x - - - -y$	$xy \sim\sim\sim -$
$abcde$	$ab*c*d*e*$
$x - (y - z)$	$xyz - -$
$x - ((y - z))$	$xyz - -$
$-(x - y) - (x + yz)w$	$xy- \sim xyz*+w*-$
$(x - y - z)(a + b - c + d)$	$xy - z - ab + c - d + *$