CSCI 1321 (Principles of Algorithm Design II), Fall 2001

Homework 8[1]

Assigned: December 4, 2001.

Due: December 11, 2001, at 5pm.

Credit: 40 points.

> *Note:* The HTML version of this document may contain hyperlinks. In this version, hyperlinks are represented by showing both the link text, formatted like this, and the full URL as a footnote.

Contents

1 Reading **1**

2 Overview **1**

3 Details **2**
3.1 Input and output specifications 2
3.2 A strategy for solving the problem 3

4 What files do I need? **3**

5 What to submit **3**

6 Hints, tips, etc. **4**
6.1 C++ programming tip: conditional compilation 4

1 Reading

Read chapter 10. Sections 7.4 and and 10.4 on tree traversals may be useful for this homework.

2 Overview

You are to convert an infix arithmetic expression to postfix notation using a tree as an intermediary data structure. You may solve the problem in any way you desire as long as it conforms to the input and output specifications, except that you may not use a stack.

[1] © 2001 Jeffrey D. Oldham (`oldham@cs.stanford.edu`) and Berna L. Massingill (`bmassing@cs.trinity.edu`). All rights reserved. This document may not be redistributed in any form without the express permission of at least one of the authors.

3 Details

3.1 Input and output specifications

Your program should read an infix arithmetic expression from the standard input and print the equivalent postfix arithmetic expression to the standard output. It should also print an error message if the input is not a well-formed infix arithmetic expression.

Infix arithmetic expressions are recursively defined: An infix expression is:

- a variable name or number, or
- something of the following form

  (*expression operator expression*)

For this assignment, we require that the five pieces of the second form (two parentheses, two expressions, and an operator) be separated by whitespace; in particular, there must be whitespace before and after the parentheses. We also define a variable, number, or operator to be any whitespace-delimited word other than "(". (This permits some fairly silly-looking "expressions", but it makes your job easier.)

Postfix arithmetic expressions are similarly defined; a postfix expression is:

- a variable name or number, or
- something of the following form

  expression expression operator

No parentheses are permitted.

Here are some examples of input and output to the function:

Input	**Output**	**Notes**
`256.46`	`256.46`	
`hello`	`hello`	
`hello xyz`	`Input expression is not well-formed.`	does not match either case of the recursive definition.
`( xyz + 45 )`	`xyz 45 +`	
`( xyz qwerty zz )`	`xyz zz qwerty`	"matches" the second case, if in a silly way.
`( x y z`	`Input expression is not well-formed.`	
`( xyz + ( abcd * 10 ) )`	`xyz abcd 10 * +`	

Reminder/hint: You can easily read a whitespace-delimited word by using the usual C++ `>>` operator and the `string` class, as in the following example:

```
string s;
cin >> s;
```

3.2 A strategy for solving the problem

Infix arithmetic expressions, postfix arithmetic expressions, trees, in-order tree traversal, and postorder tree traversal are all recursively defined. Thus, it is reasonable to think of using recursive functions to manipulate them.

Here is a possible sequence of steps to solve the problem:

1. Write a recursive function that reads an infix arithmetic expression from an `istream` and stores it in a tree (of strings). You can check that this function is working by printing the contents of the resulting tree.
2. Separately, write a recursive function that takes a tree and produces a postfix arithmetic expression.
3. Combine the two functions.

4 What files do I need?

You will (or might) need the following files:

- `tree.h`[2] contains a recursively-defined template class `Tree` as discussed in class. As with the `Seq` template class, you should not need to read the code for the `Tree` class in order to use it; see the example use program `tree-use.cpp`[3] or this short description of the class[4].
- `infix2postfix.cpp`[5] contains very partial code for the program you are to write. (That is, it's a start, but there is much to fill in.)
- `tree-debug.h`[6] contains a function `printTree()` to print the contents of a tree. (This might be useful during development/debugging.) This function takes two parameters, an `ostream` to print to (e.g., `cout`) and a `Tree` to print. To use this code:
  1. Make sure a copy of `tree-debug.h` is in your directory.
  2. Add `#include "tree-debug.h"` at the top of the file using `printTree()`.
  3. When compiling, be sure to add `-DDEBUG` as a compiler flag. See the description of conditional compilation in the section below.

  See the comments in `tree-debug.h` for additional information, and/or look at `tree-debug-use.cpp`[7] to see an example of using this code.

5 What to submit

Submit only your completed implementation, consisting of file `infix2postfix.cpp`. Submit your source code as described in the Guidelines for Programming Assignments[8]. For this assignment use a subject line of "cs1321 hw 8".

[2] `http://www.cs.trinity.edu/~bmassing/CS1321_2001fall/SamplePrograms/tree.h`
[3] `http://www.cs.trinity.edu/~bmassing/CS1321_2001fall/SamplePrograms/tree-use.cpp`
[4] `http://www.cs.trinity.edu/~bmassing/CS1321_2001fall/Notes/jdo-trees/`
[5] `http://www.cs.trinity.edu/~bmassing/CS1321_2001fall/HW08/Problems/infix2postfix.cpp`
[6] `http://www.cs.trinity.edu/~bmassing/CS1321_2001fall/HW08/Problems/tree-debug.h`
[7] `http://www.cs.trinity.edu/~bmassing/CS1321_2001fall/HW08/Problems/tree-debug-use.cpp`
[8] `http://www.cs.trinity.edu/~bmassing/CS1321_2001fall/Notes/pgmguidelines/index.html`

6 Hints, tips, etc.

6.1 C++ programming tip: conditional compilation

Frequently, programmers want to use the same C++ files to produce different executable programs. For example, when developing code, it is frequently useful to print debugging messages, but, after development finishes, these debugging messages are a nuisance. Thus, it is convenient to be able to decide at compile time whether to include or exclude the code producing the debugging messages.

File `tree-debug.h`[9], described earlier, uses the preprocessor statements `#ifdef DEBUG` and `#endif` to delimit the statements that should only be executed when debugging a program. `#ifdef DEBUG` abbreviates `#if defined(DEBUG)`, which yields true only if the preprocessor symbol `DEBUG` is defined. `#endif` marks the end of the conditional section.

To tell the compiler that the preprocessor symbol `DEBUG` should be defined and thus the debugging statements should be included in the executable, compile with the `-DDEBUG` compiler flag. For example,

```
g++ -Wall -pedantic -DDEBUG infix2postfix.cc -g -o infix2postfix
```

`-D` precedes the preprocessor symbol to define. Omitting `-DDEBUG` when compiling omits the statements between the `#ifdef` and `#endif`. `DEBUG` is not a C++ variable, function, or any other C++ thing; it is a preprocessor symbol that can only be manipulated using a `-D` compiler flag. We could just as well have used the symbol `FRED` in our code; then we would compile with (or without) the compiler flag `-DFRED`. (This "preprocessor" is a macro processor that in essence serves as a first step of the C++ compilation processor. You can read more about it in its "info" pages, using the command `info cpp`.)

[9] `http://www.cs.trinity.edu/~bmassing/CS1321_2001fall/HW08/Problems/tree-debug.h`