COMP322: Assignment 2 - Winter 2010

Due at 11:59pm EST, 3 Mar 2010

1 Introduction

For this assignment, we will actually implement a few simple classes, building on our parser from assignment 1. These additions will enable the evaluation of expressions and the assignment of values to symbols, turning your parser into a full interpreter for this small language.

2 Requirements

You will need to implement two new functional units and add them to your parser.

1. An *operand stack* to hold the numeric values of operands in the input expressions.
2. A *symbol table* to keep track of the relationship between symbols and their values.

These functional units may be made up of one or more C++ classes.

You'll use these new pieces to make your parser actually evaluate the statements that it parses. For each statement it successfully parses, the parser should print the value of the statement.

For example, suppose the input to the parser is the following three lines:

```
set a 10
set b (a/2)
1+a*b
```

Your program should print something like:

```
OK: 10
OK: 5
OK: 51
```

We'll give more details of exactly how this should work in the Hints section.

Your classes should implement enough functionality to support the necessary operations, including both a default constructor and a destructor. However, you do not need to implement a copy constructor or other features which would be needed in a "full" implementation.

Note that you should remove the display of postfix notation from your solution to this part of the assignment.

3 Operand stack

The basic purpose of the operand stack is to simplify the evaluation of parsed expressions. A stack is a fairly simple data structure which in this case will hold double-precision floating point values.

If you're not familiar with the stack data structure, it implements a "first-in, last-out" storage scheme. That is, the last value written to a stack is the first value returned by a read operation. For a full explanation, check the Wikipedia page for "stack data structure".

Your implementation should provide the following five public functions:

- `Stack(int depth=100)` - The class constructor, which will create an empty stack.
- `~Stack()` - The class destructor, which must free any allocated memory.
- `double top() const` - Return the current "top" value on the stack.
- `void push(double val)` - Place the new value on top of the stack,
- `double pop()` - Return the current top value on the stack and remove it from the stack, making the previous value the new top.

Obviously, if the stack is empty, you can't pop a value from it (this is sometimes called "stack underflow"). While this condition should not occur in your completed project, your class can simply return zero if it detects a pop from an empty stack.

Similarly, if the stack is full, you can't push a new value on it (this condition is known as "stack overflow"). You can handle this in any reasonable manner, either by growing the stack to accomodate the new value, or by discarding older values, or simply by ignoring excess push requests. If you use a large enough stack size, this error will not occur in normal operation of your interpreter.

4 Symbol table

The symbol table is the place where you store names that have had values assigned to them using the "set" statement.

You should create two classes for this part - a Symbol class, which defines the symbols themselves, and a Symtable class which is a collection of symbols.

The Symbol class can be quite simple, essentially just a C struct. It will contain the necessary fields to store a name, a value, and possibly other fields as required for the data structure you choose to implement for the symbol table.

The Symtable class will be a bit more complex. You can use any algorithm you like to organize the symbols, but the class should provide at least these four functions:

- `Symtable()` - The class constructor, which will just create an empty symbol table.
- `~Symtable()` - The class destructor. This should correctly clean up any allocated memory.
- `bool set(string name, double value)` - Add "name" to the symbol table (if it's not already there). Once the symbol exists in the table, assign it the value "value". The return code will be `true` if the operation succeeds.
- `bool get(string name, double &value)` - Look up the "name" in the symbol table, and if it is found, return its current value in the parameter "value". The return code will be `true` if the operation succeeds, and `false` if (for example) the symbol is not defined.

5 Hints

Given working implementions of the above classes, and that you have a working implementation of Assignment 1, it is fairly straightforward to incorporate them into your parser to evaluation your expressions.

Whenever you encounter a `NUMBER` or `SYMBOL`, use either `cur_val()` or `Symtable::get()` to retrieve the value associated with the constant or variable, and then push the numeric value of the token onto the operand stack.

If you encounter an undefined symbol in an expression, you should substitute some default value (e.g. zero) and print an error message.

In place of printing your postfix notation, at the points where you formerly printed an operator, you'll now pop the expected number of operands off the stack. Then you will either add, subtract, multiply, divide, or negate the operands. Then push the result back onto the stack.

For example, your `term()` function may look something like this:

```
bool term(istream &in) {
  int op;

  if (!factor(in))
    return false;

  while ((op = cur_tok()) == '*' || op == '/') {
    get_next(in);
    if (!factor(in)) {
      return false;
    }
    double y = stack.pop();
    double x = stack.pop();
    double z = // perform the correct operation...
    stack.push(z);
  }
  return true;
}
```

The implementation of the "set" statement should also be quite straightforward. When you successfully parse a "set" statement, you should note both the symbol name and the resulting value of the expression (which will be on the top of the operand stack) and use `Symtable::set` to associate the value with the symbol.

6 Submission

Submit your source files only, via email. I recommend that you create a ".h" and ".cpp" file for each functional unit. However, you can probably combine the Symbol and Symtable classes. So your final source code will probably consist of six files: `Parse.h, Parse.cpp, Symbol.h, Symbol.cpp, Stack.h`, and `Stack.cpp`

This is just a guideline, you're free to organize this some other way if you prefer. You'll still want to compile and link with the Lex.cpp file as well, of course.