COMP322: Assignment 1 - Winter 2010

Due at 11:59pm EST, 10 Feb 2010

1 Introduction

For this assignment, we'll build a parser for a very small infix expression language. For now, the code need only do the most basic checking, and convert the input into postfix notation. We need not exploit any advanced features of the C++ language.

2 Requirements

The grammar is designed to be straightforwardly converted into a recursive descent parser. You're free to use any algorithm you like to implement the parser, as long as it recognizes the language as specified.

We've provided the code for the lexical analysis phase in a file named **Lex.cpp** and **Lex.h**. You need to add the code to actually parse the high-level expression constructs.

Here are some random details:

- Your program should accept input from the "standard input" only. The standard library automatically creates an `istream` object named `std::cin` which does this.
- Your program should accept a series of statements as specified in the Grammar section below.
- Your program should demonstrate that it correctly parses the input by converting the input into postfix notation. Use 'NEG' for unary negation (to distinguish it from subtraction) and 'SET' for assignment.
- Error messages can be quite minimal. Just printing "OK" if the statement checks out, or "Syntax error" if not, is fine.

3 Grammar

Here is the grammar we need to parse, in a variant of Backus-Naur Form. Note that the '|' specifies alternation, and constructs within curly braces are repeated zero or more times.

The intent of this grammar is to allow simple 'set' expressions which assign a value to a symbol, or to simply evaluate and print an expression.

```
<statement> := <expression> '\n' | 'SET' <SYMBOL> <expression> '\n'
<expression> := <term> { '+' | '-' <term> }
<term> := <factor> { '*' | '/' <factor> }
<factor> := <SYMBOL> | <NUMBER> | '-' <factor> | '(' <expression> ')'
```

Note that symbols in quotes or all capitals are "terminals", that is, lexical constructs interpreted by the lexical analyzer. You only need to figure out how to parse the non-terminals, that is, the constructs appearing on the left-hand side of the ':=' in the rules above.

Examples of legal statements include:

```
set a (b+2) / (c+2)
a+b*c
-val
val / (-x + 10)
a*b-c*d
```

Postfix output for the above statements should be:

```
b 2 + c 2 + / a SET
a b c * +
val NEG
val x NEG 10 + /
a b * c d * -
```

4 Details on the lexical analyzer

The lexical analyzer is pretty simple (we've provided the source code) and implements four functions and some constants. You can see this in the file **Lex.h**.

```
int get_next(istream &); // Read next token
int cur_tok();           // Return current token
double cur_val();        // Return value if token is NUMBER
string cur_sym();        // Return string if token is SYMBOL
const int EOF;           // A numeric constant for end of file
const int NUMBER;        // A numeric constant, for value call cur_val()
const int SYMBOL;        // A symbol, for text call cur_sym()
const int SET_KW;        // The keyword 'set'
```

These functions and constants are all defined within a separate 'Lex' namespace. We've defined a few other things, but these aren't important right now (and may never be, we'll see).

The `get_next()` function actually does the work of reading the stream and breaking the input up into tokens. Therefore, you need to call `get_next()` before calling any of the other functions.

The other three functions simply provide access to the data found by the lexical analyzer.

The `cur_tok()` function returns the current token value. It will either be a character literal such as '+' or '/', or one of the symbolic constants. If it returns something that is not part of the language, that's an error you need to handle.

The `cur_val()` function return the current numeric value, which is only valid if the current token is `NUMBER`. You don't need it for this assignment, but we've provided it as it is trivial.

The `cur_sym()` function return the current symbol name, which is only valid if the current token is `SYMBOL`. You don't need it for this assignment, but we've provided it as it is trivial.

5 Hints

Here are some random suggestions and hints:

- Recovering after detecting an error is notoriously tricky in these sorts of parsers. You can pretty much ignore this issue, or simply read tokens until you see a newline.
- You will want to include the standard headers <string>, <iostream>, and the local header "Lex.h".
- Your code will probably be about 100-150 lines.

Here is the code for the `main()` function in our solution. Your code need not be identical:

```
int main()
{
  while (cin.good() && get_next(cin) != EOF) {
    if (!Parse::statement(cin)) {
      cout << "Error!\n";
      while (cin.good() && cur_tok() != '\n') {
        get_next(cin);
      }
    }
    else {
      cout << "OK\n";
    }
  }
}
```

5.1 Using/installing g++

g++ is the GNU C++ compiler. A recent version (4.3.2) is available on most of the Ubuntu lab computers. Our solution was developed using version 4.4.1.

We build our solution using the command line:

```
g++ -Wall -o parse Parse.cpp Lex.cpp
```

If you want to install **g++** on your system, you should be able to find instructions around the web.

For the Mac, the instructions here look reasonably accurate and up-to-date:

`http://www.edparrish.com/common/macgpp.php`

For PC/Windows systems, there are multiple choices, but the Cygwin environment mimics most of the Linux command line on Windows, so it is a good option:

`http://www.cygwin.com`

For Ubuntu (and possibly Debian) users, you can install the free g++ package with the following command:

```
sudo apt-get install g++
```

If you have trouble, or you're running some other operating system, let us know and we'll try to help you.