Phase 1: Scanning

In this first phase, you job is to implement a scanner for the SnuPL/0 language.

The input to the scanner is a character stream. The output of the scanner is a stream of tokens. The scanner must correctly recognize and tokenize keywords, identifiers, numbers, operators (assignment, binary and relational), comments, and the syntax elements of the language. The scanner should scan the input until the end of the input character stream is reached or an error is detected.

We provide a skeleton for a scanner/parser framework so that you can focus on the interesting parts. In addition, we also provide a full working example for "SnuPL/-1" that shows you how to use the framework (the EBNF for SnuPL/-1 is provided below).

The scanner skeleton can be found in the files snuplc/src/scanner.[h/cpp]. In its unmodified form, the scanner scans SnuPL/-1.

The header file first defines the token types (EToken); two corresponding data structures (ETokenName and ETokenStr) are located in the C++ file. Here, you will need to add additional tokens to implement SnuPL/0. ETokenName is used to print the token type only; ETokenStr prints the token name along with the lexeme. The elements of ETokenStr are fed to printf, so you can print the lexeme by inserting a '%s' placeholder somewhere in that string.

The token class (CToken) is fully implemented and functional, you do not have to modify it. The scanner (CScanner) is also fully functional, but only scans SnuPL/-1 in its current form. You will need to modify the function CToken* Scanner::Scan() to accept all possible tokens of SnuPL/0.

A word on the keywords: in scanner.cpp you will find a data structure called 'Keywords' which is currently empty. Put reserved keywords along with the token type in that table; CScanner then automatically initializes a list of keywords from it (see CScanner::InitKeywords). In your CScanner::Scan() method, you can then treat identifiers and keywords with the same code. After detecting an identifier, check the std::map keywords if it contains the lexeme, and if so, return the associated token type. Otherwise return an identifier.

The sources also contain a simple test program for the scanner. It creates a scanner instance and repeatedly calls CScanner::Get() to retrieve the next token until the end of file is reached. Retrieved tokens are printed to standard out.

Run

snuplc $ make test_scanner

to build it. To invoke it run it with a file name as an argument:

snuplc $ ./test_scanner ../test/scanner/test01.mod

In the directory test/scanner/ you can find a number of test files for the scanner. We advise you to create your own test cases to test special cases; we have our own set of test files to test (and grade) your scanner.

Hints: the first phase is pretty straight-forward to implement. Two points are noteworthy:

- error recovery: unrecognized lexemes should be handled by returning a tUndefined token with the illegal lexeme as its attribute
- handling of comment and whitespace: consume all whitespace and comments in the scanner (i.e., do not return a token for whitespace or comment)

Submission:

- the deadline for the first phase is **September 21, 2014 before midnight**.
- submit a tarball of your SnuPL/0 compiler by email to the TA (compiler-ta@csap.snu.ac.kr). The arrival time of your email counts as the submission time.

Do not hesitate to ask questions in class/on eTL; implementing a compiler is not an easy task. Also, start as soon as possible; if you wait until a few days before the deadline you may not be able to finish in time.

Happy coding!

Appendix: EBNF Syntax Definition of SnuPL/-1

```
module          =  statSequence ".".

digit           =  "0".."9".
number          =  digit { digit }.

factOp          =  "*" | "/"
termOp          =  "+" | "-"
relOp           =  "=" | "#"

factor          =  number | "(" expression ")".
term            =  factor { factOp factor }.
simpleexpr      =  term { termOp term }.
expression      =  simpleexpr [ relOp simplexpr ].

assignment      =  number ":=" expression.
statement       =  assignment.
statSequence    =  [ statement { ";" statement } ].

whitespace      =  { " " | \n }+.
```