Programming Exercise 2: Scanner Specification

1 Introduction

The scanner is the part of the compiler responsible for grouping characters in the input stream into tokens which are used by the parser. A typical scanner for a programming language will recognize tokens such as reserved words, identifiers, numeric constants, strings and operators.

One of the simplest ways of implementing a scanner is to use a scanner generator. There are numerous such tools available. For this exercise you will use `flex`, which stands for "Fast `lex`".

2 Using `flex`

In order to use `flex` you will need to have the `prog/gnu` module loaded. At a Unix prompt, type `module list`, and see if `prog/gnu` is listed. If it's not listed, type `module add prog/gnu` and `module initadd prog/gnu` to load the module in the current shell and the next time you log in.

Full documentation for `flex` is available as a man page. Just type `man flex` at a Unix prompt to see the documentation. You can also read the documentation for `lex`, available in the Solaris AnswerBook. There are a few differences between `lex` and `flex`, but not too many.

3 The Generated Scanner

When you compile a scanner specification using `flex`, a function named `yylex` is generated. The default definition of this function takes no arguments and returns an integer, which represents the scanned token.

Tokens are usually numbered from 257 and up, since that allows the scanner to return any single character as a token. In `scanner.l` the final rule uses this feature; any unmatched characters are returned as tokens.

The generated scanner also includes a number of important global variables and utility functions. The ones that you will encounter are the variables `yylineno`, `yyin`, `yytext` and the function `yyterminate`.

yylineno This variable holds the number of the current line of input. It is useful for error reporting, but slows the scanner down somewhat, so in order to use it, it has to be explicitly enabled using command-line options or the `yylineno` declaration in the scanner specification.

yyin This variable holds the file pointer from which `yylex` reads its input.

yytext This is a character array containing the characters that were recognized as a token.

yyterminate This function terminates the scanning process and causes `yylex` to return 0. It can be called in any action, and is the default action at the end of file.

4 The Tokens

Your scanner must to skip comments, both single-line C++ comments and multiline C style comments. If the scanner sees `/*` within a C comment it has to print a warning message. If the end of file is encountered within a C style comment, your scanner must print an error message and then terminate.

Floating-point numbers consist of an integer part followed by a period, a decimal part and an exponent. The integer and decimal parts are simply sequences of digits. The exponent part consists of the character 'E' followed by an optional sign and a sequence of digits. Either the integer or the decimal part (or both) must be given. The exponent is optional. If the integer part and exponent are both given, the decimal point and decimal part are optional. These are some valid floating-point numbers: `1.1`, `.1`, `1.`, `1E2`, `2E-3`, `.1E-4`. When your scanner recognizes a floating-point number it should return `REAL`.

Integers are simply sequences of digits that are not part of identifiers or floating-point numbers. When your scanner recognizes an integer it should return `INTEGER`.

Identifiers must start with a letter, followed by any number of digits, letters or underscore characters. When your scanner recognizes an identifier it should return `ID`.

5 Requirements

You are to finish the scanner specification in `scanner.l` by adding rules for C and C++ style comments, identifiers, integers and reals. Compile your scanner using the command `make scanner`. This generates a program named `scanner`, which you can use to test your scanner.

Run your scanner on the files in `~komp/lab2/test` and check that it generates the correct output.

Hand in the following:

- The scanner specification, with your changes clearly marked.
- Answers to the questions in the next section.
- Test data that show that the scanner works as specified.

6 Questions

Question 1 A scanner generator translates the regular expressions in the input into deterministic or nondeterministic finite automata, which is then simulated to recogize tokens in the input.

Sections 3.6–3.9 in the textbook describes how a scanner generator works. Use the techniques in 3.7 to convert your regular expression for floating-point numbers into an NFA, then use the techiques from section 3.6 to convert the NFA into a DFA. Finally use the technique described in section 3.9 to convert the regular expression directly into a DFA.

Question 2 Are the DFAs you created in question 1 minimal? If they are not, use the techniques in section 3.9 to create minimal DFAs for both automata.

7 Extra Credit Work: Regexp matcher

Implement a program that counts the number of matches of a user-supplied regular expression in a file. Optionally, implement a simple version of the Unix `grep` utility.

Your program is to convert the regular expression into an NFA or DFA (the choice is yours), which is then simulated to perform the search.

Hand in your implementation and any test data you have used.