Compiler Design
CSCI 5807

Project 1 – Lexical Analyzer Generator

Due date: TBA

Goal

Write a program that reads specifications for tokens and generates a DFA-based lexical analyzer.

Details

We will learn about deterministic finite automata (DFAs) in chapter 3.

Your program should take two parameters on the command line (maybe more): the name of the input file and the basename of the output files.

The input file contains definitions of tokens; the file specifications are given below. There are three output files that your program will generate, whose formats are also given below:

- A header file (.h) containing C++ definitions of the tokens and the analyzer class
- A source file (.cc or .cpp) containing C++ code and data for the analyzer class
- A listing file (.txt) with a human-readable form of the DFA's transition table and accepting states

Input File Specifications

The input file will consist of three types of declarations: character classes, tokens and ignore sequences. A character class is simply a set of characters that are all treated as the same (e.g., the set of uppercase and lowercase letters, when defining an identifier, are all equal). Tokens are what the DFA will search for and return. Ignore sequences are character sequences that are ignored when found; they are not returned, but neither do they generate an error.

A character class declaration has the form

class *classname* [*set*]

The *classname* can be any legal C++ name. The *set* largely follows the UNIX standard for character classes; it is a sequence of

- Characters
- Character ranges of the form α-β where α and β are characters; the range includes all characters whose ASCII / Latin-1 encodings are between α and β, including α and β.

Any whitespace between the brackets is assumed to be part of the set. If you want a hyphen to be part of the set, it should be the last character. Do not use commas or other punctuation to separate parts of the set; such punctuation is part of the set itself.

If the first character of the set is a caret ^, then the class includes all characters except those in the set. To include a caret in the set, add it anywhere other than the first character.

Use a backslash to remove any special meaning of input characters (e.g., use \] to add a right bracket to a character class.)

A token declaration has the form

token *tokenname regex*

The *tokenname* can be any legal C++ identifier. The rules for forming the regex are:

Form	**Strings it accepts**
Any character α	α
[*classname*]	Any character in the class
r_1r_2	αβ, where r_1 accepts α and r_2 accepts β
(*r*)	Any string accepted by *r*
$r_1 \vert r_2$	Any string accepted by either r_1 or r_2
*r**	Zero or more consecutive strings accepted by *r*
r+	One or more consecutive strings accepted by *r*
r?	Either the empty string or any string accepted by *r*

Closure operators (*, +, ?) have the highest precedence and are left-associative; concatenation has second-highest precedence and | has lowest precedence. Two closure operators can never be adjacent to one another.

Ignore declarations have the form

ignore *regex*

The *regex* has the same format as a token regex.

The input file can also have C++-style comments; any text beginning with // through the end of that line is to be ignored by your program. Note that if you want the pattern // to appear in either an ignore or token regex, the slashes must be escaped: \/\/

Output File Specifications

The output files declare and define a LexicalAnalyzer class in C++ and a Token enumeration. The LexicalAnalyzer class should have the following methods:

- LexicalAnalyzer(istream &input=cin)
  The constructor, taking an open istream reference that defaults to cin. Note that since ifstream is a subclass of istream, an open file stream can be used here as well.
- void start(void)
  Prepare the analyzer, or reset it if necessary / possible
- bool next(Token &t,string &lexeme) throw (invalid_argument)
  Identify the next token, if any. Returns true if a token is found, false on EOF. Passes back token and lexeme. Throws an invalid_argument exception if the input doesn't match any token or ignore sequence.

The class declaration and token enumeration should go in the header file; method definitions should go in the source file.

The listing file should show the DFA transition table in a human readable format. You are free to define whatever format works for you; however, it must show each DFA state, which characters are recognized in each state and the destination state for each character. You must also list all accepting states and which token is recognized for each accepting state.

What To Turn In

Submit your source code and output files for the sample cases I will provide online. Please send only one archive file; almost any format is okay, as long as I can extract the files.

Notes

- The output is C++ code, but you can use any language (that I can run) for your program.
- You may not use any special libraries to process regexes, as that defeats the purpose of the project.
- For some extra credit, optimize the number of states in your DFA.
- You are free to implement any algorithm for constructing the DFA. Thompson's algorithm followed by subset construction is probably easier to implement, but the regex-to-DFA algorithm described in the book might be more efficient.
- The next() method should find the longest lexeme that matches any regex. If the matched regex is an ignore pattern, then it should be ignored and the next token should be found. If the matched regex is a token, it should be returned.