Lesson 2: Introduction to Lexical Analysis

Introduction

As the first phase of a compiler, the main task of the lexical analyzer is to read the input characters of the source program, group them into lexemes, and produce as output a sequence of tokens for each lexeme in the source program. The stream of tokens is sent to the parser for syntax analysis. It is common for the lexical analyzer to interact with the symbol table as well. When the lexical analyzer discovers a lexeme constituting an identifier, it needs to enter that lexeme into the symbol table. In some cases, information regarding the kind of identifier may be read from the symbol table by the lexical analyzer to assist it in determining the proper token it must pass to the parser.

These interactions are suggested in the following figure. Commonly, the interaction is implemented by having the parser call the lexical analyzer. The call, suggested by the **getNextToken** command, causes the lexical analyzer to read characters from its input until it can identify the next lexeme and produce for it the next token, which it returns to the parser.

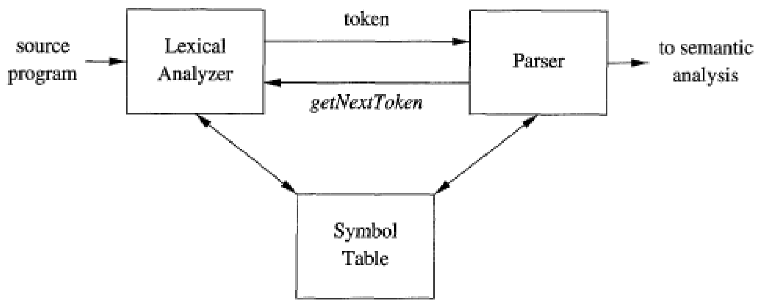

Figure 1: Interaction between Lexical Analyzer and the Parser

Since the lexical analyzer is the part of the compiler that reads the source text, it may perform certain other tasks besides identification of lexemes. One such task is stripping out comments and whitespace (blank, newline, tab, and perhaps other characters that are used to separate tokens in the input). Another task is correlating error messages generated by the compiler with the source program. For instance, the lexical analyzer may keep track of the number of newline characters seen, so it can associate a line number with each error message. In some compilers, the lexical analyzer makes a copy of the source program with the error messages inserted at the appropriate positions. If the source program uses a macro-preprocessor, the expansion of macros may also be performed by the lexical analyzer.

Sometimes, lexical analyzers are divided into a cascade of two processes:

- **Scanning** consists of the simple processes that do not require tokenization of the input, such as deletion of comments and compaction of consecutive whitespace characters into one.

- **Lexical analysis** is the more complex portion, where the scanner produces the sequence of tokens as output.

Tokens, Patterns, and Lexemes

When discussing lexical analysis, we use three related but distinct terms:

- **A token** is a pair consisting of a token name and an optional attribute value. The token name is an abstract symbol representing a kind of lexical unit, e.g., a particular keyword, or a sequence of input characters denoting an identifier. The token names are the input symbols that the parser processes.
- **A pattern** is a description of the form that the lexemes of a token may take. In the case of a keyword as a token, the pattern is just the sequence of characters that form the keyword. For identifiers and some other tokens, the pattern is a more complex structure that is matched by many strings. In all programming languages the patterns for lexemes of a token are written using **regular expressions**.
- **A lexeme** is a sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyzer as an instance of that token.

In many programming languages, the following classes cover most or all of the tokens:

- One token for each keyword
- Tokens for the operators
- One token representing all identifiers
- One or more tokens representing constants, such as numbers and literal strings.
- Tokens for each punctuation symbol, such as left and right parentheses, comma, and semicolon.

Attributes for Tokens

When more than one lexeme can match a pattern, the lexical analyzer must provide the subsequent compiler phases additional information about the particular lexeme that matched. For example, the pattern for token number matches both 0 and 1, but it is extremely important for the code generator to know which lexeme was found in the source program. Thus, in many cases the lexical analyzer returns to the parser not only a token name, but an attribute value that describes the lexeme represented by the token; the token name influences parsing decisions, while the attribute value influences translation of tokens after the parse.

Typically tokens have at most one associated attribute, although this attribute may have a structure that combines several pieces of information. The most important example is the token id, where we need to associate with the token a great deal of information. Normally, information about an identifier - e.g., its lexeme, its type, and the location at which it is first found (in case an error message about that identifier must be issued) - is kept in the symbol table. Thus, the appropriate attribute value for an identifier is a pointer to the symbol-table entry for that identifier.

Lexical Analyzer Generation

To write a lexical analyzer, you have to generate deterministic finite automations for the patterns specified using regular expressions for various token types. In addition, you have to implement sequential file reading to read the program and input buffering to match lexemes with the automatons. Given these tasks are tedious and the implementation process is the same regardless of the regular expressions, none writes lexical analyzers on his/her won. Rather, a lexical analyzer generator tool, such as LEX or FLEX, is used where you write the regular expressions and for each expression write a short piece of code (typically in C programming language) that tells what token should be sent to the syntax analyzer for any lexeme matching that expression. Note that typically, the tokens for language keywords match the regular expression for identifier (i.e., variable, function, and class names) token type. Hence, keywords would be recognized as identifiers – which we do not want. To avoid this problem, the generator tool generates the lexical analyzer in a way so that the first regular expression that matches the current lexeme is used to determine token type. Therefore, you have to write the regular expressions for keywords before that of identifier. You follow the same rules for other cases also.

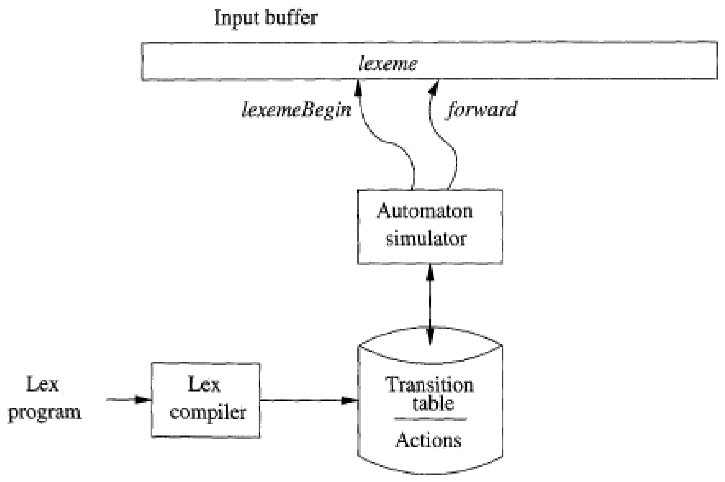

Figure 2: generation and structure of a Lexical Analyzer