School of CET

System Software and Compiler lab

Assignment No.5

TY BTech CSE

Title: Write a program using Lex specifications to implement lexical analysis phase of compiler to generate tokens of subset of Java program.

Aim: Generate lexical analyzer for Java language using LEX.

Objective:

1. To understand lexical analysis phase of compiler.
2. To understand scanner for subset of java.

Theory: Explain the following

1. Token lexeme and pattern.

Token: Token is a sequence of characters that can be treated as a single logical entity. Typical tokens are,

1)Identifiers 2) keywords 3) operators 4) special symbols 5)constants

Pattern: A set of strings in the input for which the same token is produced as output. This set of strings is described by a rule called a pattern associated with the token.

Lexeme: A lexeme is a sequence of characters in the source program that is matched by the pattern for a token.

There are some predefined rules for every lexeme to be identified as a valid token. These rules are defined by grammar rules, by means of a pattern. A pattern explains what can be a token, and these patterns are defined by means of regular expressions.

Example:

Description of token

Token	lexeme	pattern
const	const	const
if	if	if
relation	<,<=,= ,< >,>=,>	< or <= or = or < > or >= or letter followed by letters & digit
i	pi	any numeric constant
nun	3.14	any character b/w “and “except"
literal	"core"	pattern

A patter is a rule describing the set of lexemes that can represent a particular token in source program.

2. Use of regular expression (RE) in specifying lexical structure of a language.

The lexical analyzer needs to scan and identify only a finite set of valid string/token/lexeme that belong to the language in hand. It searches for the pattern defined by the language rules.

Regular expressions have the capability to express finite languages by defining a pattern for finite strings of symbols. The grammar defined by regular expressions is known as regular grammar. The language defined by regular grammar is known as regular language.

Regular expression is an important notation for specifying patterns. Each pattern matches a set of strings, so regular expressions serve as names for a set of strings. Programming language tokens can be described by regular languages. The specification of regular expressions is an example of a recursive definition. Regular languages are easy to understand and have efficient implementation.

There are a number of algebraic laws that are obeyed by regular expressions, which can be used to manipulate regular expressions into equivalent forms.

Operations

The various operations on languages are:

- Union of two languages L and M is written as
  L U M = {s | s is in L or s is in M}
- Concatenation of two languages L and M is written as
  LM = {st | s is in L and t is in M}
- The Kleene Closure of a language L is written as
  L* = Zero or more occurrence of language L.

Notations

If r and s are regular expressions denoting the languages L(r) and L(s), then

- Union : (r)|(s) is a regular expression denoting L(r) U L(s)
- Concatenation : (r)(s) is a regular expression denoting L(r)L(s)
- Kleene closure : (r)* is a regular expression denoting (L(r))*
- (r) is a regular expression denoting L(r)

Precedence and Associativity

- *, concatenation (.), and | (pipe sign) are left associative
- * has the highest precedence
- Concatenation (.) has the second highest precedence.
- | (pipe sign) has the lowest precedence of all.

Representing valid tokens of a language in regular expression

If x is a regular expression, then:

- x* means zero or more occurrence of x.
  i.e., it can generate { e, x, xx, xxx, xxxx, … }
- x+ means one or more occurrence of x.
  i.e., it can generate { x, xx, xxx, xxxx … } or x.x*
- x? means at most one occurrence of x
  i.e., it can generate either {x} or {e}.

[a-z] is all lower-case alphabets of English language.
[A-Z] is all upper-case alphabets of English language.
[0-9] is all natural digits used in mathematics.

Representing occurrence of symbols using regular expressions

letter = [a – z] or [A – Z]

digit = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 or [0-9]

sign = [+ | -]

Representing language tokens using regular expressions

Decimal = (sign)?(digit)+

Identifier = (letter)(letter | digit)*

The only problem left with the lexical analyzer is how to verify the validity of a regular expression used in specifying the patterns of keywords of a language. A well-accepted solution is to use finite automata for verification.

3. Format of lex specification file. (*.l).

Structure of a Lex Specification

A lex program consists of three parts: the definition section, the rules section, and the user subroutines.

...definition section ...

%%

... rules section ...

%%

... user subroutines ...

The parts are separated by lines consisting of two percent signs. The first two parts are required, although a part may be empty. The third part and the preceding %% line may be omitted. (This structure is the same as that used by yacc, from which it was copied.)

Definition Section

The definition section can include the literal block, definitions, internal table declarations, start conditions, and translations. (There is a section on each in this reference.) Lines that start with whitespace are copied verbatim to the C file. Typically this is used to include comments enclosed in “/*” and “*/”, preceded by whitespace.

Rules Section

The rules section contains pattern lines and C code. A line that starts with whitespace, or material enclosed in “%{” and “%}” is C code. A line that starts with anything else is a pattern line.

Alok Bhawankar - PA06 - Panel 1 - 1032170126

C code lines are copied verbatim to the generated C file. ...

Input: Subset of java language.

Output: Sequence of tokens generated by lexical analyzer and Symbol Table.

Platform: Linux (JAVA)

Conclusion: Implemented scanner in Java.
FAQs:

1. Give various tasks performed during lexical analysis phase.

   LEXICAL ANALYSIS is the very first phase in the compiler designing. A Lexer takes the modified source code which is written in the form of sentences . In other words, it helps you to convert a sequence of characters into a sequence of tokens. The lexical analyzer breaks this syntax into a series of tokens. It removes any extra space or comment written in the source code.
   Lexical analyzer performs below given tasks:

   - Helps to identify token into the symbol table
   - Removes white spaces and comments from the source program
   - Correlates error messages with the source program
   - Helps you to expands the macros if it is found in the source program
   - Read input characters from the source program

2. What is role of RE, DFA in lexical analysis?

   Regular expressions are a notation to represent lexeme patterns for a token. They are used to represent the language for lexical analyzer. They assist in finding the type of token that accounts for a particular lexeme.
   In the design of a compiler, DFA is used in the **lexical analysis** to produce tokens in the form of identifiers, keywords and constants from the input program.

3. What is LEX?
   Lex is a computer program that generates lexical analyzers ("scanners" or "lexers").

Lex is commonly used with the yacc parser generator. Lex, originally written by Mike Lesk and Eric Schmidt and described in 1975, is the standard lexical analyzer generator on many Unix systems, and an equivalent tool is specified as part of the POSIX standard

Lex reads an input stream specifying the lexical analyzer and outputs source code implementing the lexer in the C programming language. In addition to C, some old versions of Lex could also generate a lexer in Ratfor.