Lexical Analysis - Part 1

Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design
What is lexical analysis?
Why should LA be separated from syntax analysis?
Tokens, patterns, and lexemes
Difficulties in lexical analysis
Recognition of tokens - finite automata and transition diagrams
Specification of tokens - regular expressions and regular definitions
LEX - A Lexical Analyzer Generator
What is Lexical Analysis?

- The input is a high level language program, such as a ’C’ program in the form of a sequence of characters.
- The output is a sequence of tokens that is sent to the parser for syntax analysis.
- Strips off blanks, tabs, newlines, and comments from the source program.
- Keeps track of line numbers and associates error messages from various parts of a compiler with line numbers.
- Performs some preprocessor functions such as \#define and \#include in ’C’.
Separation of Lexical Analysis from Syntax Analysis

- Simplification of design - software engineering reason
- I/O issues are limited LA alone
- More compact and faster parser
  - Comments, blanks, etc., need not be handled by the parser
  - A parser is more complicated than a lexical analyzer and shrinking the grammar makes the parser faster
    - No rules for numbers, names, comments, etc., are needed in the parser
- LA based on finite automata are more efficient to implement than pushdown automata used for parsing (due to stack)
Tokens, Patterns, and Lexemes

- Running example: `float abs_zero_Kelvin = -273;`
- Token (also called *word*):
  - A string of characters which logically belong together
  - *float*, *identifier*, *equal*, *minus*, *intnum*, *semicolon*
  - Tokens are treated as terminal symbols of the grammar specifying the source language
- Pattern:
  - The set of strings for which the *same* token is produced
  - The pattern is said to *match* each string in the set
  - `float, l(l+d+_)*, =, -, d+, ;`
- Lexeme:
  - The sequence of characters matched by a pattern to form the corresponding token
  - “float”, “abs_zero_Kelvin”, “=”, “-”, “273”, “,”
Keywords, operators, identifiers (names), constants, literal strings, punctuation symbols such as parentheses, brackets, commas, semicolons, and colons, etc.

A unique integer representing the token is passed by LA to the parser.

Attributes for tokens (apart from the integer representing the token):

- **identifier**: the lexeme of the token, or a pointer into the symbol table where the lexeme is stored by the LA
- **intnum**: the value of the integer (similarly for floatnum, etc.)
- **string**: the string itself

The exact set of attributes are dependent on the compiler designer.
Difficulties in Lexical Analysis

- Certain languages do not have any reserved words, *e.g.*, `while`, `do`, `if`, `else`, etc., are reserved in 'C', but not in PL/1.

- In FORTRAN, some keywords are context-dependent.
  - In the statement, `DO 10 I = 10.86, DO10I` is an identifier, and `DO` is not a keyword.
  - But in the statement, `DO 10 I = 10, 86, DO` is a keyword.
  - Such features require substantial *look ahead* for resolution.

- Blanks are not significant in FORTRAN and can appear in the midst of identifiers, but not so in 'C'.

- LA cannot catch any significant errors except for simple errors such as, illegal symbols, etc.

- In such cases, LA skips characters in the input until a well-formed token is found.
Regular definitions, a mechanism based on regular expressions are very popular for specification of tokens
- Has been implemented in the lexical analyzer generator tool, LEX
- We study regular expressions first, and then, token specification using LEX

Transition diagrams, a variant of finite state automata, are used to implement regular definitions and to recognize tokens
- Transition diagrams are usually used to model LA before translating them to programs by hand
- LEX automatically generates optimized FSA from regular definitions
- We study FSA and their generation from regular expressions in order to understand transition diagrams and LEX
Languages

- **Symbol**: An abstract entity, not defined
  - Examples: letters and digits
- **String**: A finite sequence of juxtaposed symbols
  - $abcb$, $caba$ are strings over the symbols $a$, $b$, and $c$
  - $|w|$ is the length of the string $w$, and is the number of symbols in it
  - $\epsilon$ is the empty string and is of length 0
- **Alphabet**: A finite set of symbols
- **Language**: A set of strings of symbols from some alphabet
  - $\Phi$ and $\{\epsilon\}$ are languages
  - The set of palindromes over $\{0,1\}$ is an infinite language
  - The set of strings, $\{01, 10, 111\}$ over $\{0,1\}$ is a finite language
  - If $\Sigma$ is an alphabet, $\Sigma^*$ is the set of all strings over $\Sigma$
Language Representations

- Each subset of $\Sigma^*$ is a language
- This set of languages over $\Sigma^*$ is uncountably infinite
- Each language must have by a finite representation
  - A finite representation can be encoded by a finite string
  - Thus, each string of $\Sigma^*$ can be thought of as representing some language over the alphabet $\Sigma$
  - $\Sigma^*$ is countably infinite
  - Hence, there are more languages than language representations

- Regular expressions (type-3 or regular languages), context-free grammars (type-2 or context-free languages), context-sensitive grammars (type-1 or context-sensitive languages), and type-0 grammars are finite representations of respective languages

- $RL << CFL << CSL <<$ type-0 languages
Examples of Languages

Let $\Sigma = \{a, b, c\}$

- $L_1 = \{a^m b^n | m, n \geq 0\}$ is regular
- $L_2 = \{a^n b^n | n \geq 0\}$ is context-free but not regular
- $L_3 = \{a^n b^n c^n | n \geq 0\}$ is context-sensitive but neither regular nor context-free

Showing a language that is type-0, but none of CSL, CFL, or RL is very intricate and is omitted
Automata

- Automata are machines that accept languages
  - Finite State Automata accept RLs (corresponding to REs)
  - Pushdown Automata accept CFLs (corresponding to CFGs)
  - Linear Bounded Automata accept CSLs (corresponding to CSGs)
  - Turing Machines accept type-0 languages (corresponding to type-0 grammars)

- Applications of Automata
  - Switching circuit design
  - Lexical analyzer in a compiler
  - String processing (grep, awk), etc.
  - State charts used in object-oriented design
  - Modelling control applications, e.g., elevator operation
  - Parsers of all types
  - Compilers
Finite State Automaton

- An FSA is an **acceptor** or **recognizer** of regular languages
- An FSA is a 5-tuple, $(Q, \Sigma, \delta, q_0, F)$, where
  - $Q$ is a finite set of states
  - $\Sigma$ is the input alphabet
  - $\delta$ is the transition function, $\delta : Q \times \Sigma \rightarrow Q$
    - That is, $\delta(q, a)$ is a state for each state $q$ and input symbol $a$
  - $q_0$ is the start state
  - $F$ is the set of **final** or **accepting** states
- In one move from some state $q$, an FSA reads an input symbol, changes the state based on $\delta$, and gets ready to read the next input symbol
- An FSA **accepts** its input string, if starting from $q_0$, it consumes the entire input string, and reaches a final state
- If the last state reached is not a final state, then the input string is rejected
- $Q = \{ q_0, q_1, q_2, q_3 \}$
- $\Sigma = \{ a, b, c \}$
- $q_0$ is the start state and $F = \{ q_0, q_2 \}$
- The transition function $\delta$ is defined by the table below

<table>
<thead>
<tr>
<th>state</th>
<th>symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$a$</td>
</tr>
<tr>
<td>$q_0$</td>
<td>$q_1$</td>
</tr>
<tr>
<td>$q_1$</td>
<td>$q_1$</td>
</tr>
<tr>
<td>$q_2$</td>
<td>$q_3$</td>
</tr>
<tr>
<td>$q_3$</td>
<td>$q_3$</td>
</tr>
</tbody>
</table>

The accepted language is the set of all strings beginning with an 'a' and ending with a 'c' ($\epsilon$ is also accepted)
\[ Q = \{ q_0, q_1, q_2, q_3 \}, \text{ } q_0 \text{ is the start state} \]

\[ F = \{ q_0 \}, \delta \text{ is as in the figure} \]

Language accepted is the set of all strings of 0’s and 1’s, in which the no. of 0’s and the no. of 1’s are even numbers
The language **accepted** by an FSA is the set of all strings accepted by it, i.e., \( \delta(q_0, x) \in F \)

This is a **regular language** or a **regular set**

Later we will define **regular expressions** and **regular grammars** which are **generators** of regular languages

It can be shown that for every regular expression, an FSA can be constructed and vice-versa
Nondeterministic FSA

- NFAs are FSA which allow 0, 1, or more transitions from a state on a given input symbol.
- An NFA is a 5-tuple as before, but the transition function $\delta$ is different.
  $$\delta(q, a) = \text{the set of all states } p, \text{ such that there is a transition labelled } a \text{ from } q \text{ to } p$$
- $\delta : Q \times \Sigma \rightarrow 2^Q$
- A string is accepted by an NFA if there exists a sequence of transitions corresponding to the string, that leads from the start state to some final state.
- Every NFA can be converted to an equivalent deterministic FA (DFA), that accepts the same language as the NFA.
Nondeterministic FSA Example - 1

L = \{ x | x contains two consecutive 0's or two consecutive 1's \}

<table>
<thead>
<tr>
<th>state</th>
<th>symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>q0</td>
<td>{q_0, q_3}</td>
</tr>
<tr>
<td>q1</td>
<td>\phi</td>
</tr>
<tr>
<td>q2</td>
<td>{q_2}</td>
</tr>
<tr>
<td>q3</td>
<td>{q_4}</td>
</tr>
<tr>
<td>q4</td>
<td>{q_4}</td>
</tr>
</tbody>
</table>