CS143 P2 Design Document

Shantanu Phadke

October 2021

1 Overview

A Lexical Analyzer is the first of the three "frontend" phases of a compiler. It breaks up an input program into valid tokens (otherwise known as lexemes) and also assigns the appropriate Token class names for each of the tokens. This information is then passed onto the next phase of the compiler, the parser.

In order to implement a lexical analyzer for COOL in JAVA, a tool called JLex, which is an example of a Lexical Analyzer Generator, has been utilized. The main JLex definitions for the Lexical Analyzer have been coded in cool.lex, and this has then been made into the JAVA class CoolLexer.java.

2 Key Files

• **test.cl** - Sample test file for the lexical analyzer that has some lexical issues.
• **cool.lex** - Main file with JFlex implementation of the COOL Lexical Analyzer.

3 Key Data Structures

3.1 Abstract Table

Typically a given input program will have many instances of the same lexemes. In order to save space and time, these lexemes are usually stored in a Table object. In our case, there is a provided base class called AbstractTable, that is extended by two sub-classes: (i) StringTable and (ii) IntTable.

As one can expect, the StringTable is used to store values of strings (like identifiers or literal string values) that are encountered in the inputted program, where as int values in the inputted program are stored in the IntTable as their literal strings.

3.2 Finite Automata

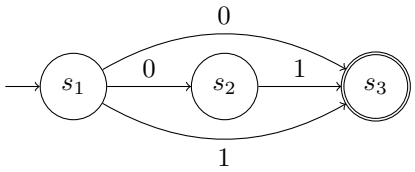
A Finite Automata is a Data Structure that consists of two main components:: (i) states, and (ii) state transitions. States can be visualized as nodes while state transitions can be viewed as edges between different states. There are two special types of states: (i) start states, and (ii) accepting states. Start states have no incoming edges while accepting states have no outgoing edges. Typically, a Finite Automata will have a single start state but multiple accepting states.

Our Lexical Analyzer is essentially a Finite Automata consisting of its own states and state transitions. There are two types of Finite Automata: (i) Nondeterministic Finite Automata or NFAs, and (ii) Deterministic Finite Automata or DFAs, both of which are discussed in greater detail below.

While most of the work relating to Converting NFAs to DFAs and implementing DFAs as tables is done by JLex behind-the-scenes, it is still important to understand the concept of NFAs/DFAs when coming up with our Lexical Specification in cool.lex.

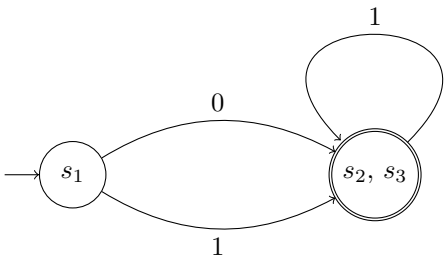
3.2.1 Nondeterministic Finite Automata (NFA)

Nondeterministic Finite Automatas (NFAs), can have none, one or multiple state transitions for a given input, per state. Below is an example of a NFA that accepts the strings '0', '1' and '01':



3.2.2 Deterministic Finite Automata (DFA)

Deterministic Finite Automatas, called DFAs for short-form, can have only a single state transition for a given input per state. Below is an example of the DFA equivalent of the above NFA:



4 Lexical Analyzer Implementation (JLex)

The JLex file cool.lex consists of a few distinct sections. The three main sections are: (i) User Code, (ii) JLex Directives, and (iii) Regular Expression Rules.

4.1 User Code

User code is simply code that gets copied into the JAVA Lexer class that JLex will generate. In our case there is only one line here that imports JCup's Symbol class.

4.2 JLex Directives

In our case, the JLex Directives section of the JLex file contains 4 main components: (i) Initialization Code for the Lexical Analyzer class, (ii) End of File Code for the Lexical Analyzer Class, (iii) Macro Definitions, and (iv) State Definitions.

4.2.1 Initialization Code

This is code we want to include at the top of our Lexical Analyzer to instantiate any key variables that it will use. In the case of our COOL Lexical Analyzer, the following table highlights the main variables that are used. These were all provided with the starter code:

Variable	Purpose
MAX_STR_CONST	Specifies max. length of Strings in COOL
string_buf	Keeps track of current String constant being read
curr_lineno	Tracks current line number being processed
filename	Holds the name of the current file being processed

4.2.2 End of File Code

This is code that specifies the way we want our Lexical Analyzer to behave when it reaches the End of the inputted program's file in various states. The following table summarizes how our COOL Lexical Analyzer will behave if it encounters EOF in its various states:

State	EOF Behavior
YYINITIAL	Return the EOF Symbol
MULTI_LINE_COMMENT	Go back to the initial state and return an Error with the message "EOF in Comment"
STRING	Go back to the initial state and return an Error with the message "EOF in String Constant"

4.2.3 Macro Definitions

Macro Definitions are simply variable initializations that our Lexical Analyzer will use. The following table provides an overview of the main Macro Definitions that have been defined in cool.lex, as well as their purposes:

Name	Definition
DIGIT	Digits from 0 to 9
INTEGER	Digits repeated one or more times
CAPITAL	Capital letters A through Z
LOWER	Lowercase lettters z through z
LETTER	A Lowercase or Capital letter
TYPE_IDENTIFIER	Digits from 0 to 9
OBJECT_IDENTIFIER	Digits repeated one or more times
WHITESPACE	Capital letters A through Z
STRING	Lowercase lettters z through z
SINGLELINE_COMMENT_START	- -
MULTILINE_COMMENT_START	(*
MULTILINE_COMMENT_END	*)
CLASS	[Cc][Ll][Aa][Ss][Ss]
ELSE	[Ee][Ll][Ss][Ee]
FALSE	[f][Aa][Ll][Ss][Ee]
FI	[Ff][Ii]
IF	[Ii][Ff]
IN	[Ii][Nn]
INHERITS	[Ii][Nn][Hh][Ee][Rr][Ii][Tt][Ss]
ISVOID	[Ii][Ss][Vv][Oo][Ii][Dd]
LET	[Ll][Ee][Tt]
LOOP	[Ll][Oo][Oo][Pp]
POOL	[Pp][Oo][Oo][Ll]
THEN	[Tt][Hh][Ee][Nn]
WHILE	[Ww][Hh][Ii][Ll][Ee]
CASE	[Cc][Aa][Ss][Ee]
ESAC	[Ee][Ss][Aa][Cc]
NEW	[Nn][Ee][Ww]
OF	[Oo][Ff]
NOT	[Nn][Oo][Tt]
TRUE	[t][Rr][Uu][Ee]

4.2.4 State Definitions

As described in the NFA and DFA sections, Lexical Analyzers use the notion of state to keep track of how exactly they should process new lexemes. The following table provides an overview of our Lexical Analyzer's main state definitions,

as well as what each signifies:

State	Purpose
YYINITIAL	The initial state of the Lexical Analyzer
SINGLE_LINE_COMMENT	The state for processing single-line comments
MULTI_LINE_COMMENT	The state for processing multi-line comments
STRING	The state for processing string constants

4.3 Regular Expression Rules

This is the core section of our Lexical Analyzer, and provides instructions for what to do when various token classes are encountered in different states. The following table provides a brief overview of what our Lexical Analyzer does in some of these cases:

State + Input(s)	Result
YYINITIAL + TYPE_IDENTIFIER	Add entry to stringtable and return TYPEID Symbol
YYINITIAL + OBJECT_IDENTIFIER	Add entry to stringtable and return OBJECTID Symbol
YYINITIAL + INTEGER	Add entry to inttable and return INT_CONST Symbol
YYINITIAL + " (Opening)	Change state to STRING and empty string_buf
YYINITIAL + NEWLINE	Increment curr_lineno
YYINITIAL + SINGLELINE_COMMENT_START	Change state to SINGLELINE_COMMENT
YYINITIAL + MULTILINE_COMMENT_START	Change state to MULTILINE_COMMENT
YYINITIAL + All Other Inputs	Return appropriate Symbol
STRING + " (Closing)	Change state to YYINITIAL and add string_buf to stringtable
STRING + NULL CHARACTER	Return error
STRING + NEWLINE	Return error
STRING + All Other Inputs	Add to string_buf
SINGLE_LINE_COMMENT + NEWLINE	Increment curr_lineno and change state to YYINITIAL
SINGLE_LINE_COMMENT + All Other Inputs	Do Nothing
MULTI_LINE_COMMENT + NEWLINE	Increment curr_lineno