Page:
Date: 11

Assignment B2

Title: Lexical Analysis - Token Generation

Date of Completion: 11 - 02 - 2020

Problem Statement:
Write a program using LEX specifications to implement lexical analysis phase of compiler to generate tokens of subset of Java Program.

Objectives:
1. To implement lexical analysis phase of compiler
2. To understand LEX specifications
3. To generate tokens of subject of Java program

Outcome:
Students will be able to:
1. Implement lexical analysis phase of compiler.
2. Understand LEX specifications
3. Analyse tokens of a Java Program

Software & Hardware Requirements:
Fedora 64 bit OS
Eclipse IDE for java
i5 Processor
4GB RAM
500 GB HDD
GCC compiler
lex compiler

Page:
Date:

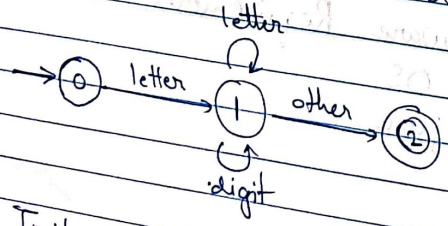
Theory:

During the first phase compiler reads the input & converts the string in the source to tokens. With regular expression we can specify patterns to lex so it can generate code that will allow it to scan & match strings in the input. Each pattern specified in the input to lex has an associated action. Typically an action requires a token that represents the matched string for subsequent uses by the parser. Initially we will simply print the matched string rather than return a token value.

The following represents a simple pattern, comprised of a regular expression that scans for identifiers. Lex will read this pattern & produce C code for a lexical analyzer that scans for identifiers.

letter (letter | digit) *

This pattern matches a string of characters that begin with a single letter followed by zero / more letters & digits.



Finite State Automata

Page:
Date: / /

Regular expressions are translated by lex to a computer program that mimics an FSA. Using the next input character & current state the next state is easily determined by indexing into a computer generated state table.

Lex's Limitation:

Lex cannot recognise nested structures such as parenthesis. To handle the same, stacks are incorporated. When (is encountered it is pushed onto stack, which is popped when) is encountered.

YACC augments FSA with a stack to parse nested structures with ease.

Lex is good at pattern matching while YACC is appropriate for more daunting tasks.

Pattern	Matches
.	Any character except newline
\.	Literal .
\n	Newline
\t	tab
^	beginning of line
$	end of line

Name	Function
int yylex(void)	Call to invoke lexer, returns token
char *yytext	Pointer to matched string
yyleng	Length of matched string
yylval	Value associated with token
int yywrap(void)	Wrap Up, return 1 if done, 0 if not.
FILE *yyout	Output file
FILE *yyin	Input file
INITIAL	Initial start condition
BEGIN	Condition switch start condition
ECHO	Write matched string.

Regular expression are used for pattern matching.
Character class defines a single characters.
Operators lose their meaning in character class.
Hyphen (-) & Circumvent (^) are 2 operators allowed
- represents a range of characters.
^ negates an expression.

```
... definitions ...
%%
... rules ...
%%
... subroutines ...
%%
```

Input to lex is divided into 3 sections
%% separates the 3 sections
There must always be the rule section.

Page :
Date : / /

Test Cases :

Description	Input	Output	Result
Preprocessor	import java.io.*;	Preprocessor	Success
Access Specifiers	public class Input	public - access specifier	Success
Parenthesis	if (a == 13)	(- Parenthesis begin) - Parenthesis end	Success
Datatype	int a;	int - datatype	Success
End of line	a = 12;	; - delimeter	Success
Equal to sign	a = 12;	= - assignment op	Success
Relational operator	if (a == 13)	== - relational op	Success
Identifier	a = 12;	a - identifier	Success
Constant Value	a = 12;	12 - constant integer	Success.

Conclusion :

We have sucessfully implemented lexical analysis phase of a compiler to generate tokens of a Java Program.

Sneha Bhor