## Java Language Scanner - Printing Token Set

### Introduction

A scanner (or lexical analyzer) reads input code and breaks it into tokens. For a Java language scanner, tokens include keywords, identifiers, operators, literals, and punctuation. The scanner outputs the token set, including the Token Class, Token Value, and Line Number.

### Token Set Format

The format for each token in the output is as follows:

Token Class: <class> Token Value: <value> Line Number: <line>

**Example:**

* Token Class: keyword Token Value: "for" Line Number: 2
* Token Class: id1 Token Value: "i" Line Number: 2

### **Using Flex to Create the Scanner**

Flex is a tool for generating scanners. The typical structure of a Flex file for a Java scanner is:

%{

#include <stdio.h>

int line\_num = 1;

%}

%%

\n { line\_num++; }

[0-9]+ { printf("Token Class: num Token Value: \"%s\" Line Number: %d\n", yytext, line\_num); }

[a-zA-Z\_][a-zA-Z0-9\_]\* { printf("Token Class: id%d Token Value:

%%

int main() {

yylex();

return 0;

}

### Understanding Flex

Flex (Fast Lexical Analyzer) is a tool used to generate scanners (lexical analyzers). It reads a set of rules and patterns defined by the user and produces a C program that scans for these patterns in the input text. The generated program, typically called lex.yy.c, can then be compiled with GCC.

**Flex File Structure**

A Flex file generally consists of three sections, separated by %%:

1. **Definition Section:**

* Enclosed between %{ and %}.
* Used for including C headers and defining global variables or functions.

1. **Rules Section:**

* Defined after the first %%.
* Each rule consists of a pattern (using regular expressions) and an action (typically a C code block to handle the matched text).
* **Example:**

[a-zA-Z\_][a-zA-Z0-9\_]\* { printf("Identifier: %s\n", yytext); }

1. **User Code Section:**

* Defined after the second %%.
* Contains the main() function and any additional user-defined functions.

**Key Points:**

* Flex automatically generates the yylex() function that matches patterns from the input and executes the associated action.
* The yytext variable holds the matched text.
* The yylineno variable provides the current line number.

**Compiling the Scanner using GCC and Flex**

To compile and run the scanner:

1. Generate the scanner C file:

* flex scanner.l

1. Compile with GCC:

* gcc lex.yy.c -o scanner -lfl

1. Run the scanner with input:

* ./scanner < input.java

**Additional GCC Commands**

1. To create an executable:

* gcc -o myprogram myprogram.c

1. To generate assembly code:

* gcc -S myprogram.c

1. To link and produce an executable:

* gcc -o output file1.o file2.o

### Conclusion

This scanner effectively identifies Java language tokens, printing the token class, value, and line number. Using Flex and GCC ensures efficient compilation and execution of the scanner program.

**CFG for Compiler (Java)**

**Generic Rules**

**Statement Rules**

<statement\_list> ::= <statement> | <expression> <statement\_list>

<statement> ::= <expression\_statement> | <return\_statement> | <if\_statement> | <for\_loop>

**Expression Rules (Arithmetic & Comparison)**

<expression\_statement> ::= <expression> “;”

<expression\_opt> ::= <expression> | ε

<expression> ::= <identifier> | <number> | <expression> <operator> <expression> | <function\_call> | “(“ <expression> ”)”

**Return**

<return\_statement> ::= “return” <expression\_opt> “;”

**Type**

<type> ::= "int" | "float" | "char" | "bool" | “void”

**Operator**

<operator> ::= "+" | "-" | "\*" | "/" | “=” | “<” | “<=” | “>” | “>=” | “==” | “!=” | “!”

**Identifier**

<identifier> ::= [a-zA-Z\_][a-zA-Z0-9\_]\*

**Number**

<number> ::= [0-9]+

**Arrays**

<array\_declaration> ::= <type> <identifier> "[" <size> "]"

<array\_access> ::= <identifier> "[" <index> "]"

<size> ::= <number>

<index> ::= <expression>

**Functions**

<function\_definition> ::= <type><identifiers> “(“ <parameter\_list\_opt “)” “{“ <statement\_list> “}”

<parameter\_list\_opt> ::= <parameter\_list> | ε

<parameter\_list> ::= <parameter> | <parameter> “,” <parameter\_list>

<parameter> ::= <type><identifier>

<function\_call> ::= <identifier> “(“ <argument\_list\_opt> “)”

<argument\_list\_opt> ::= <argument\_list> | ε

<argument\_list> ::= <expression> | <expression> “,” <argument\_list>

**If-Else Logic**

<if\_statement> ::= “if” “(“ <expression> “)” “{“ <statement\_list> “}” | “if” “(“ <expression> “)” “{“ <statement\_list> ”}” “else” “{“ <statement\_list> “}”

**For-Loop Logic**

<for\_loop> ::= “for” “(“ <init>; <expression>; <update> “)” “{“ <statement\_list> ”}”

<init> ::= <type> <identifier> “=” <expression>

<update> ::= <identifier>“++” | <identifier>>”--” | <identifier> ”+=” <expression> | <identifier> ”-=” <expression>