COMPLIER DESIGN MINI PROJECT

“PARSER FOR JAVA”

SUBMITTED BY:

K K VISHNU (ROLLNO 2)

T N ANSHUMANTH RAO (ROLL NO 3)

SHREEKANTH N SHENOY (ROLL NO 4)

**OBJECTIVE:**

To design a parser for JAVA grammar using a top down or bottom up approach by using any available programming language.

**Grammar:**

The grammar to be used in BNF form is:

compilation\_unit =

[ package\_statement ]

< import\_statement >

< type\_declaration > .

package\_statement =

"package" package\_name ";" .

import\_statement =

"import" ( ( package\_name "." "\*" ";" )

/ ( class\_name / interface\_name ) ) ";" .

type\_declaration =

[ doc\_comment ] ( class\_declaration / interface\_declaration ) ";" .

doc\_comment = "/\*\*" "... text ..." "\*/" .

class\_declaration =

< modifier > "class" identifier

[ "extends" class\_name ]

[ "implements" interface\_name < "," interface\_name > ]

"{" < field\_declaration > "}" .

interface\_declaration =

< modifier > "interface" identifier

[ "extends" interface\_name < "," interface\_name > ]

"{" < field\_declaration > "}" .

field\_declaration =

( [ doc\_comment ] ( method\_declaration

/ constructor\_declaration

/ variable\_declaration ) )

/ static\_initializer

/ ";" .

method\_declaration =

< modifier > type identifier

"(" [ parameter\_list ] ")" < "[" "]" >

( statement\_block / ";" ) .

constructor\_declaration =

< modifier > identifier "(" [ parameter\_list ] ")"

statement\_block .

statement\_block = "{" < statement > "}" .

variable\_declaration =

< modifier > type variable\_declarator

< "," variable\_declarator > ";" .

variable\_declarator =

identifier < "[" "]" > [ "=" variable\_initializer ] .

variable\_initializer =

expression

/ ( "{" [ variable\_initializer

< "," variable\_initializer > [ "," ] ] "}" ) .

static\_initializer =

"static" statement\_block .

parameter\_list =

parameter < "," parameter > .

parameter =

type identifier < "[" "]" > .

statement =

variable\_declaration

/ ( expression ";" )

/ ( statement\_block )

/ ( if\_statement )

/ ( do\_statement )

/ ( while\_statement )

/ ( for\_statement )

/ ( try\_statement )

/ ( switch\_statement )

/ ( "synchronized" "(" expression ")" statement )

/ ( "return" [ expression ] ";" )

/ ( "throw" expression ";" )

/ ( identifier ":" statement )

/ ( "break" [ identifier ] ";" )

/ ( "continue" [ identifier ] ";" )

/ ( ";" ) .

if\_statement =

"if" "(" expression ")" statement

[ "else" statement ] .

do\_statement =

"do" statement "while" "(" expression ")" ";" .

while\_statement =

"while" "(" expression ")" statement .

for\_statement =

"for" "(" ( variable\_declaration / ( expression ";" ) / ";" )

[ expression ] ";"

[ expression ] ";"

")" statement .

try\_statement =

"try" statement

< "catch" "(" parameter ")" statement >

[ "finally" statement ] .

switch\_statement =

"switch" "(" expression ")" "{"

< ( "case" expression ":" )

/ ( "default" ":" )

/ statement >

"}" .

expression =

numeric\_expression

/ testing\_expression

/ logical\_expression

/ string\_expression

/ bit\_expression

/ casting\_expression

/ creating\_expression

/ literal\_expression

/ "null"

/ "super"

/ "this"

/ identifier

/ ( "(" expression ")" )

/ ( expression

( ( "(" [ arglist ] ")" )

/ ( "[" expression "]" )

/ ( "." expression )

/ ( "," expression )

/ ( "instanceof" ( class\_name / interface\_name ) )

) ) .

numeric\_expression =

( ( "-"

/ "++"

/ "--" )

expression )

/ ( expression

( "++"

/ "--" ) )

/ ( expression

( "+"

/ "+="

/ "-"

/ "-="

/ "\*"

/ "\*="

/ "/"

/ "/="

/ "%"

/ "%=" )

expression ) .

testing\_expression =

( expression

( ">"

/ "<"

/ ">="

/ "<="

/ "=="

/ "!=" )

expression ) .

logical\_expression =

( "!" expression )

/ ( expression

( "ampersand"

/ "ampersand="

/ "|"

/ "|="

/ "^"

/ "^="

/ ( "ampersand" "ampersand" )

/ "||="

/ "%"

/ "%=" )

expression )

/ ( expression "?" expression ":" expression )

/ "true"

/ "false" .

string\_expression = ( expression

( "+"

/ "+=" )

expression ) .

bit\_expression =

( "~" expression )

/ ( expression

( ">>="

/ "<<"

/ ">>"

/ ">>>" )

expression ) .

casting\_expression =

"(" type ")" expression .

creating\_expression =

"new" ( ( classe\_name "(" [ arglist ] ")" )

/ ( type\_specifier [ "[" expression "]" ] < "[" "]" > )

/ ( "(" expression ")" ) ) .

literal\_expression =

integer\_literal

/ float\_literal

/ string

/ character .

arglist =

expression < "," expression > .

type =

type\_specifier < "[" "]" > .

type\_specifier =

"boolean"

/ "byte"

/ "char"

/ "short"

/ "int"

/ "float"

/ "long"

/ "double"

/ class\_name

/ interface\_name .

modifier =

"public"

/ "private"

/ "protected"

/ "static"

/ "final"

/ "native"

/ "synchronized"

/ "abstract"

/ "threadsafe"

/ "transient" .

package\_name =

identifier

/ ( package\_name "." identifier ) .

class\_name =

identifier

/ ( package\_name "." identifier ) .

interface\_name =

identifier

/ ( package\_name "." identifier ) .

integer\_literal =

( ( "1..9" < "0..9" > )

/ < "0..7" >

/ ( "0" "x" "0..9a..f" < "0..9a..f" > ) )

[ "l" ] .

float\_literal =

( decimal\_digits "." [ decimal\_digits ] [ exponent\_part ] [ float\_type\_suffix ] )

/ ( "." decimal\_digits [ exponent\_part ] [ float\_type\_suffix ] )

/ ( decimal\_digits [ exponent\_part ] [ float\_type\_suffix ] ) .

decimal\_digits =

"0..9" < "0..9" > .

exponent\_part =

"e" [ "+" / "-" ] decimal\_digits .

float\_type\_suffix =

"f" / "d" .

character =

"based on the unicode character set" .

string =

"''" < character > "''" .

identifier =

"a..z,$,\_" < "a..z,$,\_,0..9,unicode character over 00C0" >

**PROGRAMMING LANGUAGES USED:**

* **Bison**: It is a general-purpose parser generator that converts a grammar description for an LALR (1) context-free grammar into a C program to parse that grammar. The Bison parser is a bottom-up parser. It tries, by shifts and reductions, to reduce the entire input down to a single grouping whose symbol is the grammar's start-symbol. We use it to see if the given grammar could be parsed.
* **FLEX**: It is a free and open-source software alternative to Lex. It is a computer program that generates lexical analyzers. We use it to generate tokens.
* **C**: It is a general-purpose, imperative computer programming language, supporting structured programming, lexical variable scope and many more. We use it to implement symbol table and hashing.

**TYPE OF PARSER:**

We use Bottom-up Parsing techniques as we are using Bison to implement parsing. Bottom-up parsing recognizes the text's lowest-level small details first, before its mid-level structures, and leaving the highest-level overall structure to last.

The bottom-up name comes from the concept of a parse tree, in which the most detailed parts are at the bottom of the (upside-down) tree, and larger structures composed from them are in successively higher layers, until at the top or "root" of the tree a single unit describes the entire input stream. A bottom-up parse discovers and processes that tree starting from the bottom left end, and incrementally works its way upwards and rightwards. A parser may act on the structure hierarchy's low, mid, and highest levels without ever creating an actual data tree; the tree is then merely implicit in the parser's actions. Bottom-up parsing lazily waits until it has scanned and parsed all parts of some construct before committing to what the combined construct is.

**METHEDOLOGY:**

The project is comprised of a flex program that takes the input file and generates a stream of tokens from it. It does this by matching the input characters to rules specified in the program. This part acts as the lexical analyzer. The bison program receives the sequence of tokens from the lexical analyzer and recognizes its structure. Using shifts (moving the input symbol onto the stack) and reductions (replacing a substring on the top of the stack by the LHS of the appropriate rule that generates this string), the parser reduces the entire input into a single symbol, in the case of a successful parse. The bison program contains all the grammar rules that define the language.

For this project, we began with the Backus Normal Form (BNF) rules for the Java program. BNF is a metasyntactic notation procedure that can be used to specify the syntax of computer programming languages. Using these rules, we wrote the grammar for Java in the format required for Bison. Following this, we had a list of tokens expected by the program, and the Flex file was created to generate and return all these tokens.

**USER DOCUMENTATION:**

Run the following script on the terminal, the parsing action/verdict will be shown in the terminal, the symbol table will be stored in an output file.

//Script

bison -d bison\_file.y

flex flex\_file.l

gcc lex.yy.c bison\_file.tab.c -o op1

./op1

flex symbol.l

gcc lex.yy.c -o op2

./op2

**CODE:**

**//FLEX CODE**

%{

#include <stdio.h>

#include <stdlib.h>

#include "bison\_file.tab.h"

%}

%%

"\t"|" "|"\n" {printf("%s",yytext);}

(("#".\*)|("//".\*))"\n" { }

("/\*"(.|"\n")\*"\*/") { return DOC\_COMMENT; }

"\"".\*"\"" {printf("%s",yytext);return LITERAL;}

"\'"."\'" | "\'"\n"\'" {printf("%s",yytext);return SINGLE\_QUOTE;}

[0-9]+ {printf("%s",yytext); return NUM;}

"FLOAT\_TYPE" {printf("%s",yytext); return FLOAT\_TYPE\_SUFFIX;}

"[" {printf("%s",yytext); return LSB;}

"]" {printf("%s",yytext); return RSB;}

"{" {printf("%s",yytext); return LC;}

"}" {printf("%s",yytext); return RC;}

"(" {printf("%s",yytext); return LP;}

")" {printf("%s",yytext); return RP;}

"," {printf("%s",yytext); return COMMA;}

";" {printf("%s",yytext); return SCOLON;}

":" {printf("%s",yytext); return COLON;}

"." {printf("%s",yytext); return DOT;}

"?" {printf("%s",yytext); return QUESMARK;}

"+" {printf("%s",yytext); return ADD;}

"-" {printf("%s",yytext); return SUB;}

"\*" {printf("%s",yytext); return MUL;}

"/" {printf("%s",yytext); return DIV;}

"%" {printf("%s",yytext); return MOD;}

"=" {printf("%s",yytext); return ASSIGN;}

"+=" {printf("%s",yytext); return ADDASSIGN;}

"-=" {printf("%s",yytext); return SUBASSIGN;}

"\*=" {printf("%s",yytext); return MULASSIGN;}

"/=" {printf("%s",yytext); return DIVASSIGN;}

"%=" {printf("%s",yytext); return MODASSIGN;}

"++" {printf("%s",yytext); return INCREMENT;}

"--" {printf("%s",yytext); return DECREMENT;}

"EXPONENT" {printf("%s",yytext); return EXP;}

"\"" {printf("%s",yytext); return DQ;}

"<" {printf("%s",yytext); return LT;}

">" {printf("%s",yytext); return GT;}

"<=" {printf("%s",yytext); return LTE;}

">=" {printf("%s",yytext); return GTE;}

"==" {printf("%s",yytext); return EQEQ;}

"!=" {printf("%s",yytext); return NTEQ;}

"!" {printf("%s",yytext); return NOT;}

"&" {printf("%s",yytext); return BAND;}

"|" {printf("%s",yytext); return BOR;}

"&=" {printf("%s",yytext); return BANDEQ;}

"|=" {printf("%s",yytext); return BOREQ;}

"^" {printf("%s",yytext); return BXOR;}

"^=" {printf("%s",yytext); return BXOREQ;}

"&&" {printf("%s",yytext); return LAND;}

"||" {printf("%s",yytext); return LOR;}

"~" {printf("%s",yytext); return TILDE;}

">>" {printf("%s",yytext); return SIGNED\_RSHIFT;}

"<<" {printf("%s",yytext); return SIGNED\_LSHIFT;}

">>>" {printf("%s",yytext); return UNSIGNED\_RSHIFT;}

"true" {printf("%s",yytext); return TRUE;}

"false" {printf("%s",yytext); return FALSE;}

"abstract" {printf("%s",yytext); return ABSTRACT;}

"assert" {printf("%s",yytext); return ASSERT;}

"boolean" {printf("%s",yytext); return BOOLEAN;}

"break" {printf("%s",yytext); return BREAK;}

"byte" {printf("%s",yytext); return BYTE;}

"case" {printf("%s",yytext); return CASE;}

"catch" {printf("%s",yytext); return CATCH;}

"char" {printf("%s",yytext); return CHAR;}

"class" {printf("%s",yytext); return CLASS;}

"const" {printf("%s",yytext); return CONST;}

"continue" {printf("%s",yytext); return CONTINUE;}

"default" {printf("%s",yytext); return DEFAULT;}

"do" {printf("%s",yytext); return DO;}

"double" {printf("%s",yytext); return DOUBLE;}

"else" {printf("%s",yytext); return ELSE;}

"enum" {printf("%s",yytext); return ENUM;}

"extends" {printf("%s",yytext); return EXTENDS;}

"final" {printf("%s",yytext); return FINAL;}

"finally" {printf("%s",yytext); return FINALLY;}

"float" {printf("%s",yytext); return FLOAT;}

"for" {printf("%s",yytext); return FOR;}

"goto" {printf("%s",yytext); return GOTO;}

"if" {printf("%s",yytext); return IF;}

"implements" {printf("%s",yytext); return IMPLEMENTS;}

"import" {printf("%s",yytext); return IMPORT;}

"instanceof" {printf("%s",yytext); return INSTANCEOF;}

"int" {printf("%s",yytext); return INT;}

"interface" {printf("%s",yytext); return INTERFACE;}

"long" {printf("%s",yytext); return LONG;}

"native" {printf("%s",yytext); return NATIVE;}

"new" {printf("%s",yytext); return NEW;}

"package" {printf("%s",yytext); return PACKAGE;}

"private" {printf("%s",yytext); return PRIVATE;}

"protected" {printf("%s",yytext); return PROTECTED;}

"public" {printf("%s",yytext); return PUBLIC;}

"return" {printf("%s",yytext); return RETURN;}

"short" {printf("%s",yytext); return SHORT;}

"static" {printf("%s",yytext); return STATIC;}

"strictfp" {printf("%s",yytext); return STRICTFP;}

"super" {printf("%s",yytext); return SUPER;}

"switch" {printf("%s",yytext); return SWITCH;}

"synchronized" {printf("%s",yytext); return SYNCHRONIZED;}

"this" {printf("%s",yytext); return THIS;}

"threadsafe" {printf("%s",yytext); return THREADSAFE;}

"throw" {printf("%s",yytext); return THROW;}

"throws" {printf("%s",yytext); return THROWS;}

"transient" {printf("%s",yytext); return TRANSIENT;}

"try" {printf("%s",yytext); return TRY;}

"void" {printf("%s",yytext); return VOID;}

"volatile" {printf("%s",yytext); return VOLATILE;}

"while" {printf("%s",yytext); return WHILE;}

"String" {printf("%s",yytext); return STRING;}

"0x"[0-9a-f]+ {printf("%s",yytext); return HEXNUM;}

"redundant" {printf("%s",yytext); return REDUNDANT;}

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

%%

int yywrap(){return 1;}

//**BISON CODE**

%{

#include<stdio.h>

#include<stdlib.h>

#define YYDEBUG 1

int yylex();

int yyerror();

extern FILE \*yyin;

#

%}

%token ABSTRACT ASSERT BOOLEAN BREAK BYTE CASE CATCH CHAR CLASS CONST CONTINUE DEFAULT DO DOUBLE ELSE ENUM EXTENDS FINAL FINALLY FLOAT FOR GOTO IF IMPLEMENTS IMPORT INSTANCEOF INT INTERFACE LONG NATIVE NEW PACKAGE PRIVATE PROTECTED PUBLIC RETURN SHORT STATIC STRICTFP SUPER SWITCH SYNCHRONIZED THIS THREADSAFE THROW THROWS TRANSIENT TRY VOID VOLATILE WHILE SCOLON DOT LC RC LP RP LSB RSB COLON QUESMARK COMMA ADD SUB MUL DIV MOD ASSIGN ADDASSIGN SUBASSIGN MULASSIGN DIVASSIGN MODASSIGN INCREMENT DECREMENT EXP DQ LT LTE GT GTE EQEQ NTEQ TRUE FALSE NUM NOT BAND BOR BANDEQ BOREQ BXOR BXOREQ LAND LOR TILDE SIGNED\_RSHIFT UNSIGNED\_RSHIFT SIGNED\_LSHIFT FLOAT\_TYPE\_SUFFIX HEXNUM IDENTIFIER LITERAL STRING REDUNDANT DOC\_COMMENT

%%

compilation\_unit : package\_statement import\_statement\_more type\_declaration\_more

| import\_statement\_more type\_declaration\_more ;

import\_statement\_more : import\_statement import\_statement\_more

| eps ;

type\_declaration\_more : type\_declaration type\_declaration\_more

| eps ;

package\_statement : PACKAGE package\_name SCOLON ;

import\_statement : IMPORT package\_name DOT MUL SCOLON

| IMPORT class\_name SCOLON

| IMPORT interface\_name SCOLON ;

type\_declaration : DOC\_COMMENT class\_declaration

| DOC\_COMMENT interface\_declaration

| class\_declaration

| interface\_declaration ;

class\_declaration : modifier\_more CLASS IDENTIFIER class\_declaration\_1 class\_declaration\_2 LC field\_declaration\_more RC

modifier\_more : modifier modifier\_more

| eps ;

field\_declaration\_more : field\_declaration field\_declaration\_more

| eps ;

class\_declaration\_1 : EXTENDS class\_name

| eps ;

class\_declaration\_2 : IMPLEMENTS interface\_name interface\_name\_more

| eps ;

interface\_name\_more : COMMA interface\_name interface\_name\_more

| eps ;

interface\_declaration : modifier\_more INTERFACE IDENTIFIER interface\_declaration\_1 LC field\_declaration\_more RC ;

interface\_declaration\_1 : EXTENDS interface\_name\_more

| eps ;

field\_declaration : method\_declaration

| constructor\_declaration

| variable\_declaration

| DOC\_COMMENT method\_declaration

| DOC\_COMMENT constructor\_declaration

| DOC\_COMMENT variable\_declaration

| static\_initializer

| SCOLON ;

method\_declaration : modifier\_more type IDENTIFIER LP parameter\_list RP square\_brak\_more method\_declaration\_1;

method\_declaration\_1 : statement\_block

| SCOLON ;

square\_brak\_more : LSB RSB square\_brak\_more

| LSB expression RSB square\_brak\_more

| eps ;

constructor\_declaration : modifier\_more IDENTIFIER LP parameter\_list RP statement\_block ;

statement\_block : LC statement\_more RC ;

statement\_more : statement statement\_more

| eps ;

variable\_declaration : modifier\_more type variable\_declarator variable\_declaration\_1 SCOLON

| IDENTIFIER IDENTIFIER SCOLON

| IDENTIFIER IDENTIFIER square\_brak\_more variable\_declaration\_1 SCOLON

| IDENTIFIER IDENTIFIER square\_brak\_more ASSIGN expression SCOLON;

variable\_declaration\_1 : COMMA variable\_declarator variable\_declaration\_1

| eps ;

variable\_declarator : IDENTIFIER square\_brak\_more variable\_declarator\_1 ;

variable\_declarator\_1 : ASSIGN variable\_initializer

| eps ;

variable\_initializer : expression

| LC variable\_initializer\_1 RC ;

variable\_initializer\_1 : variable\_initializer

| variable\_initializer COMMA

| variable\_initializer variable\_initializer\_more

| variable\_initializer variable\_initializer\_more COMMA

| eps ;

variable\_initializer\_more : COMMA variable\_initializer variable\_initializer\_more

| eps ;

static\_initializer : STATIC statement\_block ;

parameter\_list : parameter parameter\_more

| eps ;

parameter\_more : COMMA parameter parameter\_more

| eps ;

parameter : type IDENTIFIER square\_brak\_more ;

statement : variable\_declaration

| expression SCOLON

| statement\_block

| if\_statement

| do\_statement

| while\_statement

| for\_statement

| try\_statement

| switch\_statement

| SYNCHRONIZED LP expression RP statement

| RETURN SCOLON

| RETURN expression SCOLON

| THROW expression SCOLON

| IDENTIFIER COLON statement

| BREAK SCOLON

| BREAK IDENTIFIER SCOLON

| CONTINUE SCOLON

| CONTINUE IDENTIFIER SCOLON

| SCOLON ;

if\_statement : IF LP expression RP statement

| IF LP expression RP statement ELSE statement ;

do\_statement : DO statement WHILE LP expression RP SCOLON ;

while\_statement : WHILE LP expression RP statement ;

for\_statement : FOR LP for\_statement\_1 RP statement ;

for\_statement\_1 : for\_statement\_1\_1 for\_statement\_1\_2 SCOLON for\_statement\_1\_2 ;

for\_statement\_1\_1 : variable\_declaration

| expression SCOLON

| SCOLON ;

for\_statement\_1\_2 : expression

| eps ;

try\_statement : TRY statement try\_statement\_1

| TRY statement try\_statement\_1 FINALLY statement ;

try\_statement\_1 : CATCH LP parameter RP statement try\_statement\_1

| eps;

switch\_statement : SWITCH LP expression RP LC switch\_statement\_1 RC ;

switch\_statement\_1 : CASE expression COLON switch\_statement\_1

| DEFAULT COLON switch\_statement\_1

| statement switch\_statement\_1

| eps ;

expression : numeric\_expression

| testing\_expression

| logical\_expression

| bit\_expression

| casting\_expression

| creating\_expression

| literal\_expression

| SUPER

| THIS

| IDENTIFIER

| LP expression RP

| expression expression\_1 ;

expression\_1 : LP RP

| LP arglist RP

| LSB expression RSB

| DOT expression

| COMMA expression

| INSTANCEOF class\_name

| INSTANCEOF interface\_name;

numeric\_expression : SUB expression

| INCREMENT expression

| DECREMENT expression

| expression INCREMENT

| expression DECREMENT

| expression ADD expression

| expression SUB expression

| expression MUL expression

| expression DIV expression

| expression MOD expression

| expression ASSIGN expression

| expression ADDASSIGN expression

| expression SUBASSIGN expression

| expression DIVASSIGN expression

| expression MULASSIGN expression

| expression MODASSIGN expression ;

testing\_expression : expression LT expression

| expression GT expression

| expression LTE expression

| expression GTE expression

| expression EQEQ expression

| expression NTEQ expression ;

logical\_expression : NOT expression

| expression BAND expression

| expression BOR expression

| expression BANDEQ expression

| expression BOREQ expression

| expression BXOR expression

| expression BXOREQ expression

| expression LAND expression

| expression LOR expression

| expression QUESMARK expression COLON expression

| TRUE

| FALSE ;

bit\_expression : TILDE expression

| expression SIGNED\_RSHIFT expression

| expression SIGNED\_LSHIFT expression

| expression UNSIGNED\_RSHIFT expression ;

casting\_expression : LP type RP expression ;

creating\_expression : NEW class\_name LP RP

| NEW class\_name LP arglist RP

| NEW type\_specifier

| NEW type\_specifier LSB expression RSB

| NEW type\_specifier square\_brak\_more

| NEW LP expression RP ;

literal\_expression : integer\_literal

| float\_literal

| string

;

arglist : expression arglist\_1 ;

arglist\_1 : COMMA expression arglist\_1

| eps ;

type : type\_specifier square\_brak\_more ;

type\_specifier : BOOLEAN

| BYTE

| CHAR

| SHORT

| INT

| FLOAT

| LONG

| DOUBLE

| STRING

| VOID

|IDENTIFIER

;

modifier : PUBLIC

| PRIVATE

| PROTECTED

| STATIC

| FINAL

| NATIVE

| SYNCHRONIZED

| ABSTRACT

| TRANSIENT

| THREADSAFE ;

package\_name : IDENTIFIER

| package\_name DOT IDENTIFIER ;

class\_name : IDENTIFIER

| package\_name DOT IDENTIFIER ;

interface\_name : IDENTIFIER

| package\_name DOT IDENTIFIER ;

integer\_literal : NUM

| HEXNUM

float\_literal : NUM DOT

| NUM DOT NUM

| NUM DOT exponent\_part

| NUM DOT NUM exponent\_part

| NUM DOT NUM FLOAT\_TYPE\_SUFFIX

| NUM DOT exponent\_part FLOAT\_TYPE\_SUFFIX

| NUM DOT NUM exponent\_part FLOAT\_TYPE\_SUFFIX

| DOT NUM

| DOT NUM exponent\_part

| DOT NUM FLOAT\_TYPE\_SUFFIX

| DOT NUM exponent\_part FLOAT\_TYPE\_SUFFIX

| NUM exponent\_part

| NUM FLOAT\_TYPE\_SUFFIX

| NUM exponent\_part FLOAT\_TYPE\_SUFFIX ;

exponent\_part : EXP NUM

| EXP ADD NUM

| EXP SUB NUM

string : LITERAL ;

eps : ;

%%

int yyerror(char \*msg){

printf("\nInvalid expression");

return 1;

}

void main (){

/\*#ifdef YYDEBUG

yydebug = 1;

#endif\*/

yyin=fopen("inpfile.java","r");

do{

if(yyparse() )

{

printf("\n Failure!");

exit(0);

}

}while(!feof(yyin));

printf("\n\nSuccessfully parsed!!!\n");

}

//**SYMBOL TABLE HEADER FILE**

struct token{

int hashVal;

char name[256];

char type[256];

int size;

char scope;

int arg;

int args[50];

char retType[256];

struct token\* next;

};

typedef struct token\* TOKEN;

TOKEN table[20];

void initialize\_symtable(){

for(int i=0;i<20;i++)

table[i] = NULL;

}

int compHash(char \*str){

int hashVal = 0;

for(int i=0;i<strlen(str);i++)

hashVal += (i+1)\*(int)str[i];

hashVal %= 20;

//printf("\nHashVal of %s = %d", str, hashVal);

return hashVal;

}

int searchVal(char name[]){

int hash = compHash(name);

TOKEN check = table[hash];

while(check != NULL){

if(strcmp(name, check->name) == 0)

return hash;

check = check->next;

}

return -1;

}

void insertToken(char name[], char type[], int size, char scope, int arg, int args[], char retType[]){

//printf("\nInside Insert Funtion for %s\n", name);

if(searchVal(name) != -1){

return;

}

TOKEN temp = (TOKEN)malloc(sizeof(struct token));

//printf("\nMem Alocated...\n");

temp->hashVal = compHash(name);

strcpy(temp->name, name);

strcpy(temp->type, type);

temp->size = size;

temp->scope = scope;

temp->arg = arg;

for(int i=0;i<arg;i++)

temp->args[i] = args[i];

if(retType == NULL){

strcpy(temp->retType, " ");

}

else{

strcpy(temp->retType, retType);

}

//printf("Copied items into temp.\n");

temp->next = NULL;

//printf("Copied items into temp.\n");

if(table[temp->hashVal] == NULL)

table[temp->hashVal] = temp;

else{

TOKEN check = table[temp->hashVal];

while(check->next != NULL)

check = check->next;

check->next = temp;

}

//printf("Done\n");

}

void displaySymTab(){

FILE \*fp;

fp = fopen("sym\_out.txt", "w");

//printf("\n ----------------------------------------------------------------------------");

fprintf(fp, "\nHash | ID Name | ID Type | Size | Scope | Arg | Arg Vals | Ret Type |");

fprintf(fp, "\n---------------------------------------------------------------------");

for(int i=0;i<20;i++){

if(table[i] != NULL){

TOKEN check = table[i];

while(check != NULL){

fprintf(fp, "\n%3d | %30s | %8s | %5d | %5c | %5d | ", check->hashVal, check->name, check->type, check->size, check->scope, check->arg);

for(int j=0;j<check->arg;j++)

fprintf(fp, "%d", check->args[j]);

fprintf(fp," \t\t\t|%s\t\t\t", check->retType);

check = check->next;

}

}

}

//printf("\n ----------------------------------------------------------------------------");

fclose(fp);

}

**//SYMBOL TABLE FLEX CODE**

%{

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include "hash\_symtab.h"

int checkKeyword(char buf[]){

char \*keys[] = {"abstract" , "assert" , "boolean" , "break" , "byte" , "case" , "catch" ,

"char" , "class" , "const" , "continue" , "default" , "do" , "double" , "else" , "enum" , "extends" ,

"final" , "finally" , "float" , "for" , "goto" , "if" , "implements" , "import" , "instanceof" , "int" ,

"interface" , "long" , "native" , "new" , "package" , "private" , "protected" , "public" , "return" ,

"short" , "static" , "strictfp" , "super" , "switch" , "synchronized" , "this" , "threadsafe" , "throw" ,

"throws" , "transient" , "try" , "void" , "volatile" , "while" , "scolon" , "dot" , "lc" , "rc" , "lp" ,

"rp" , "lsb" , "rsb" , "colon" , "quesmark" , "comma" , "add" , "sub" , "mul" , "div" , "mod" , "assign" ,

"addassign" , "subassign" , "mulassign" , "divassign" , "modassign" , "increment" , "decrement" , "exp" ,

"dq" , "lt" , "lte" , "gt" , "gte" , "eqeq" , "nteq" , "true" , "false" , "num" , "not" , "band" , "bor" ,

"bandeq" , "boreq" , "bxor" , "bxoreq" , "land" , "lor" , "tilde" , "signed\_rshift" , "unsigned\_rshift" ,

"signed\_lshift" , "float\_type\_suffix" , "hexnum" , "identifier" , "literal" , "String" , "redundant" ,

"doc\_comment"};

for(int i=0;i<105;i++){

if((strcmp(buf, keys[i]) == 0 ))

return 1;

}

return 0;

}

char global\_type[256], global\_scope = 'G';

int is\_func = 0, size,c=0;

int argc = 0, argVals[105];

char buf[256], func\_name[256], func\_retType[256], scope;

%}

%%

"import".\*"\n" {

}

"/\*"(.|"\n")\*"\*/" {

}

"//".\*"\n" {

}

"\""(.)\*"\"" {

}

"{" {

c++;

if(c!=0)

global\_scope = 'L';

else

global\_scope = 'R';

}

"}" { c--;

if(c!=0)

global\_scope = 'L';

else

global\_scope = 'R';

}

")" {

if(is\_func == 1){

insertToken(func\_name, "FUNC", 0, scope, argc, argVals, func\_retType);

for(int i=0;i<50;i++){

argVals[i] = 0;

}

argc = 0;

is\_func = 0;

}

}

[a-zA-Z][a-zA-Z0-9\_]\*[a-zA-Z0-9]\* {

int j = checkKeyword(yytext);

if(j == 0){

if(strcmp(global\_type, "int") == 0 || strcmp(global\_type, "float") == 0){

size = 4;

}

else if(strcmp(global\_type, "char") == 0){

size = 1;

}

else if(strcmp(global\_type, "double") == 0){

size = 8;

}

if(is\_func == 1){

argVals[argc] = compHash(yytext);

argc++;

}

insertToken(yytext, global\_type, size, global\_scope, 0, NULL, " ");

}

else{

strcpy(global\_type, yytext);

}

}

[a-zA-Z][a-zA-Z0-9\_]\*[a-zA-Z0-9]\*("["[0-9]+"]") {

//printf("\nIntel Inside..%s\n", yytext);

int i=0;

char buf[256];

while(yytext[i] != '['){

buf[i] = yytext[i];

i++;

}

buf[i] = '\0';

i++;

int num = 0;

while(yytext[i]!=']'){

num = num\*10 + ((int)yytext[i]-48);

i++;

}

if(strcmp(global\_type, "int") == 0 || strcmp(global\_type, "float") == 0){

size = 4;

}

else if(strcmp(global\_type, "char") == 0){

size = 1;

}

else if(strcmp(global\_type, "double") == 0){

size = 8;

}

if(is\_func == 1){

argVals[argc] = compHash(buf);

argc++;

}

insertToken(buf, global\_type, size\*num, global\_scope, 0, NULL, " ");

}

[a-zA-Z][a-zA-Z0-9\_]\*[a-zA-Z0-9]\*"(" {

int i=0;

while(yytext[i] != '('){

buf[i] = yytext[i];

i++;

}

buf[i] = '\0';

int j = checkKeyword(buf);

if(j == 0){

strcpy(func\_name, buf);

is\_func = 1;

strcpy(func\_retType, global\_type);

scope = global\_scope;

argc = 0;

}

}

(.|"\n") {}

%%

int yywrap(){

return 1;

}

int main(){

initialize\_symtable();

yyin = fopen("inpfile.java", "r");

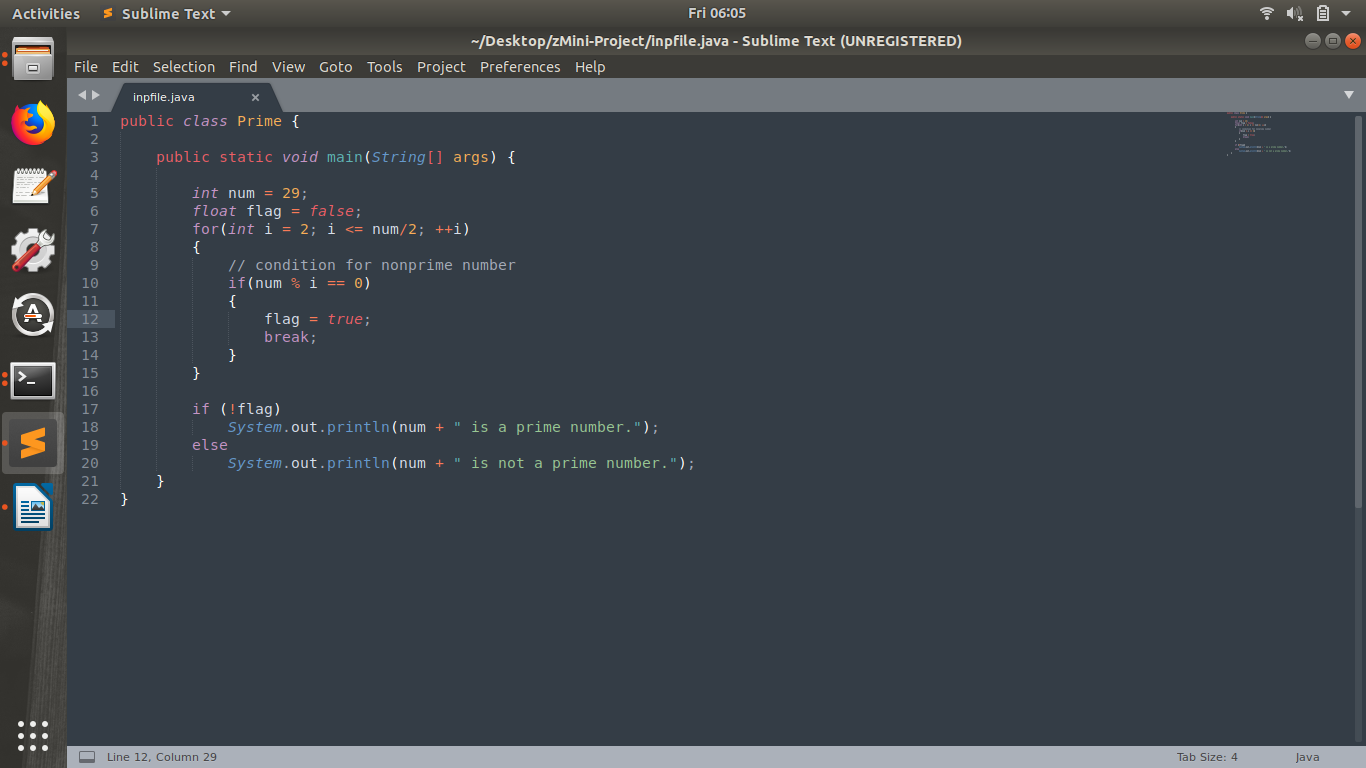
yylex();

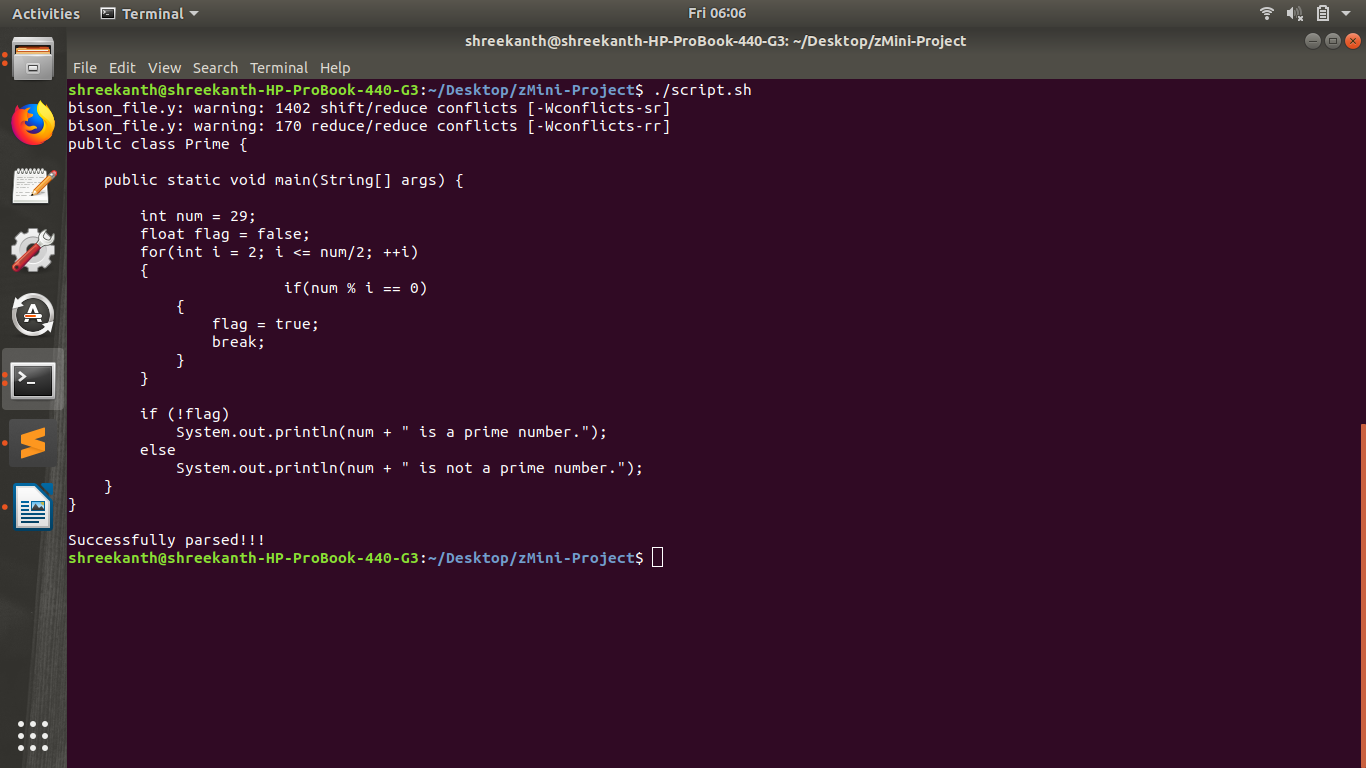
displaySymTab();

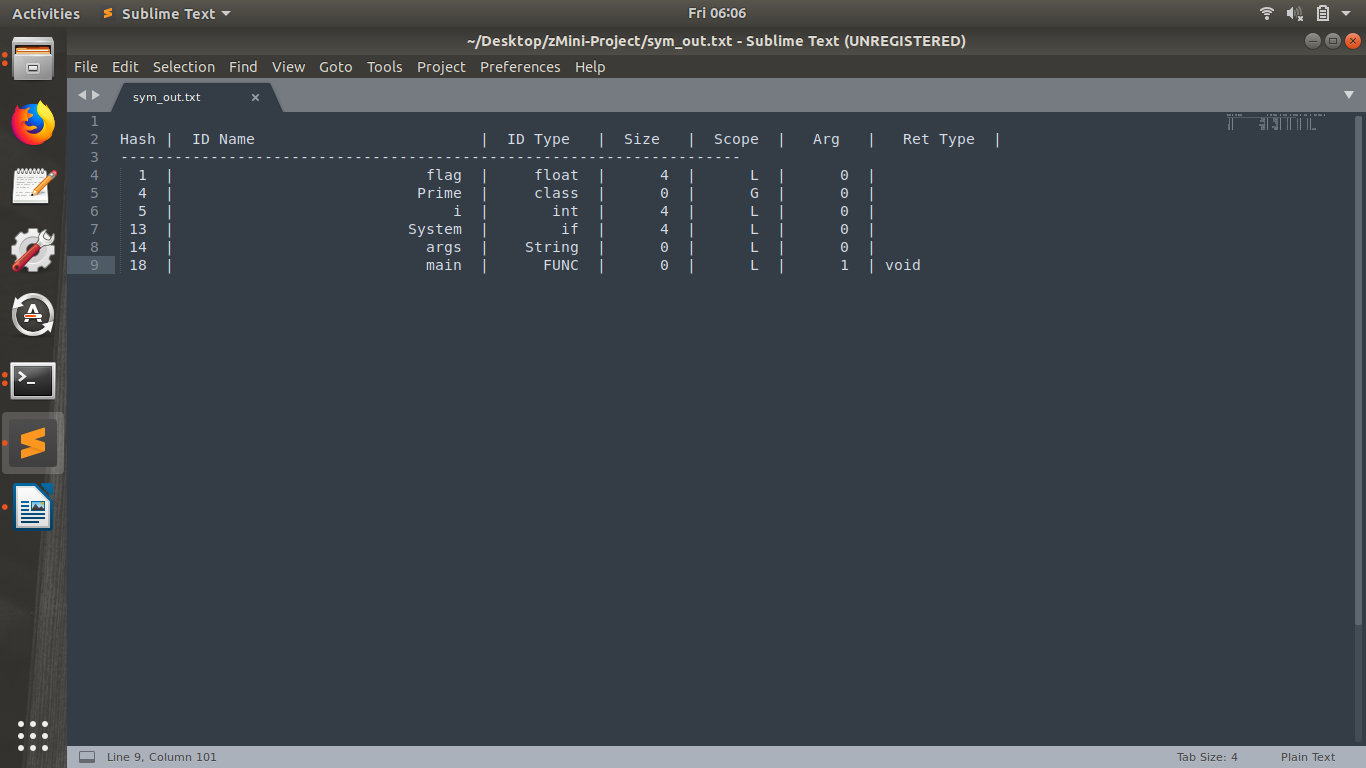
}

**CODE SNAPSHOTS:**

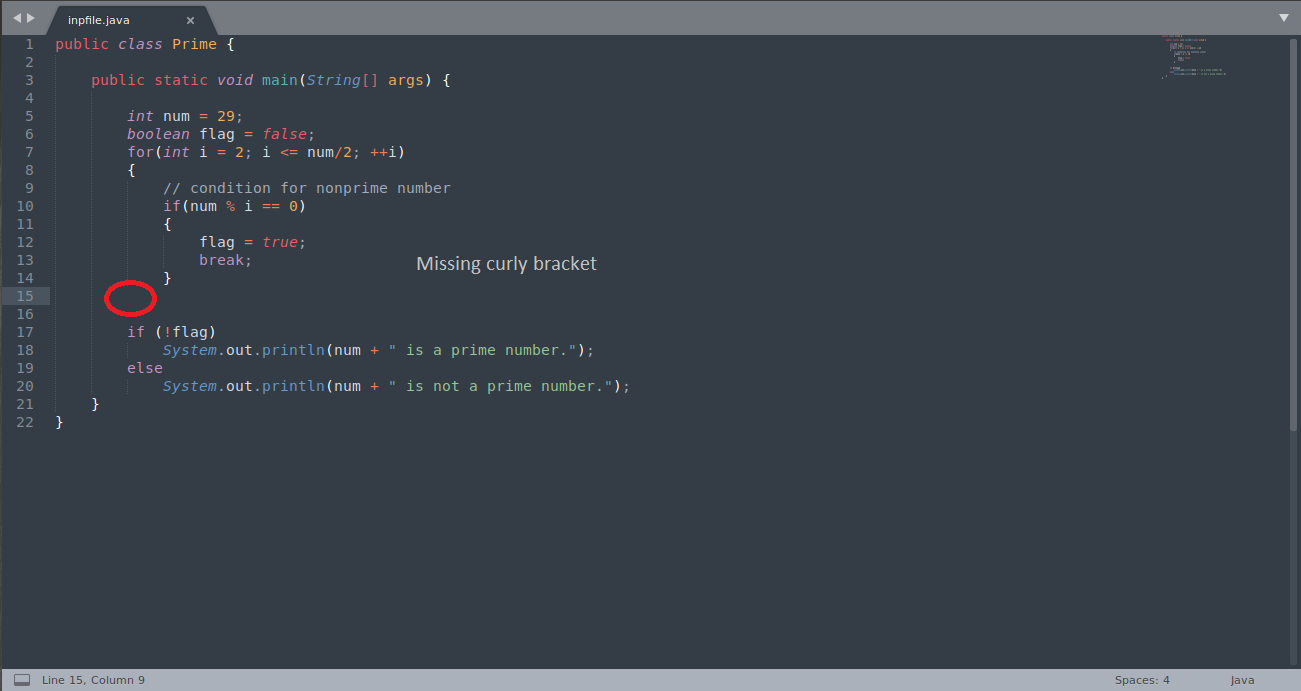
**//TEST CASE 1**

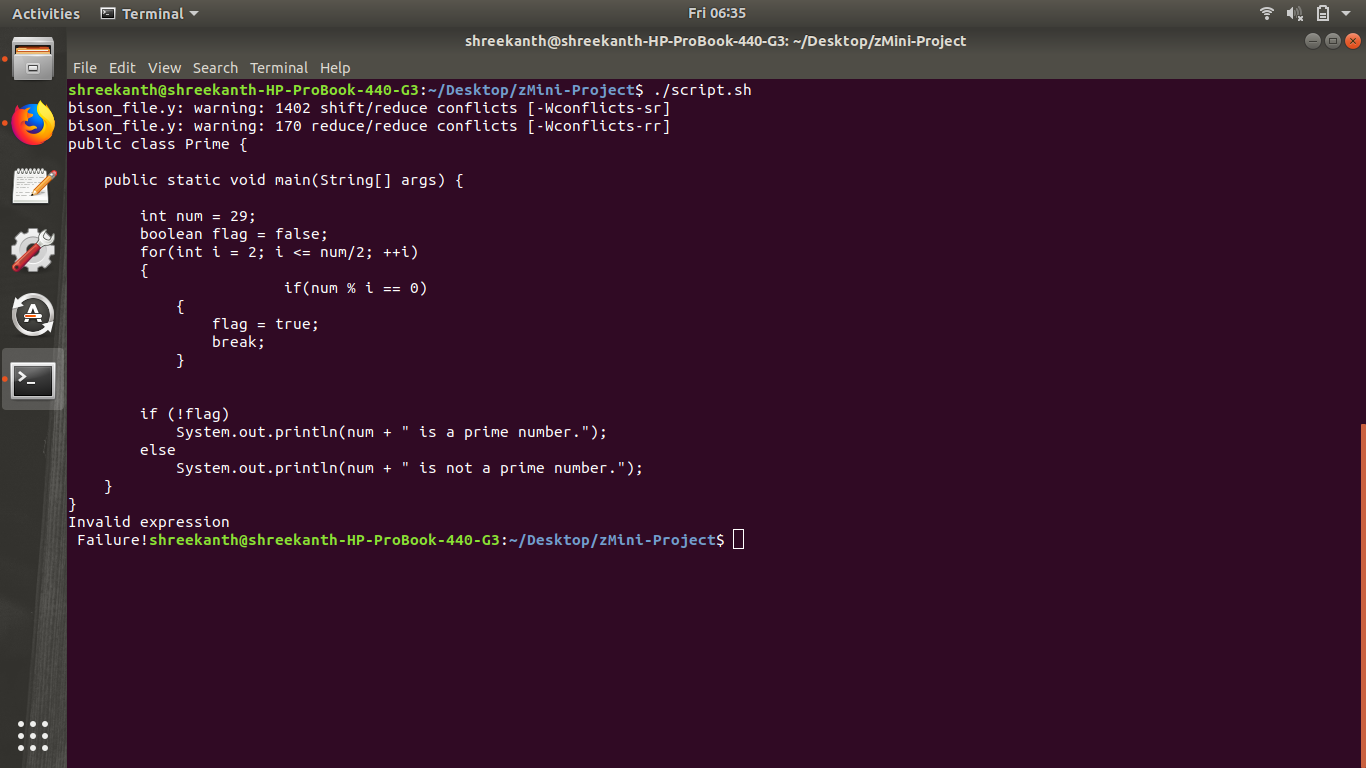
 Input:

 Terminal Output:

Test File Output

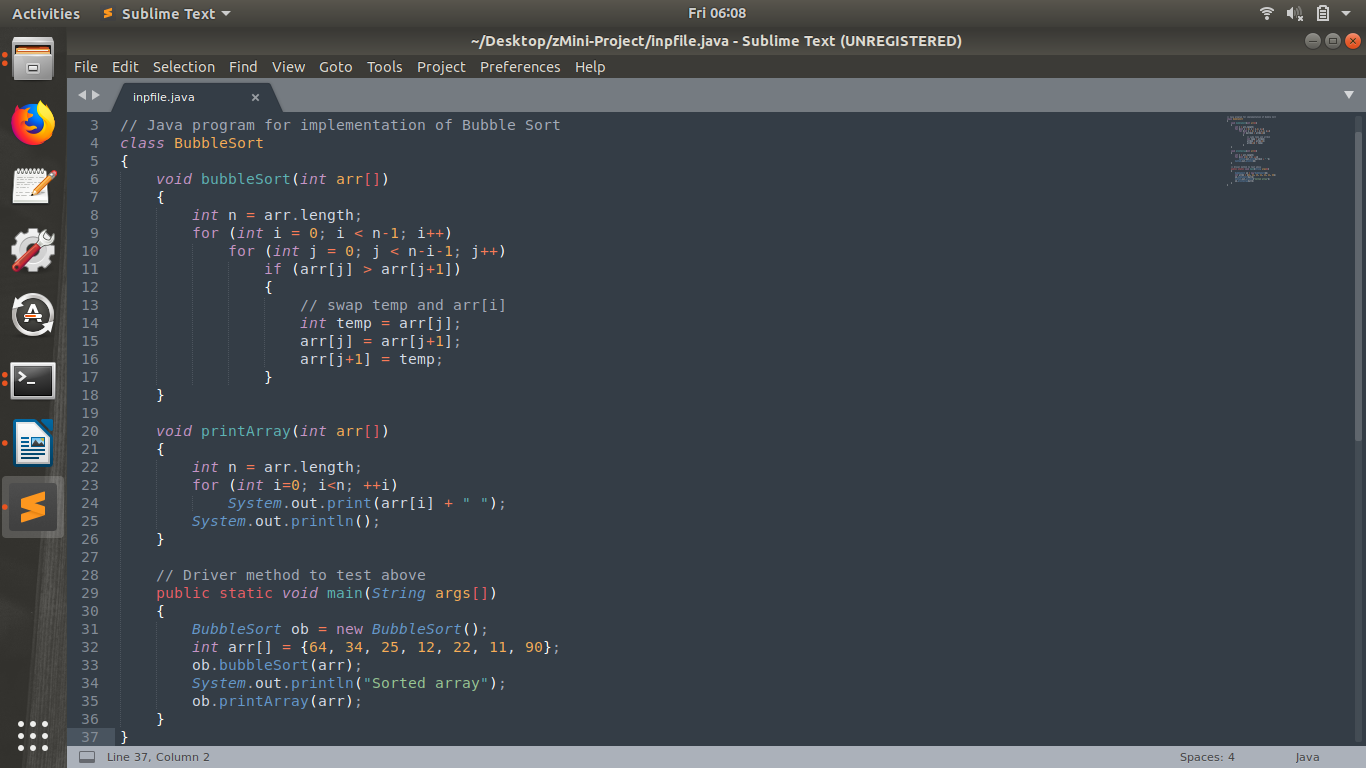
Error Induced Input:



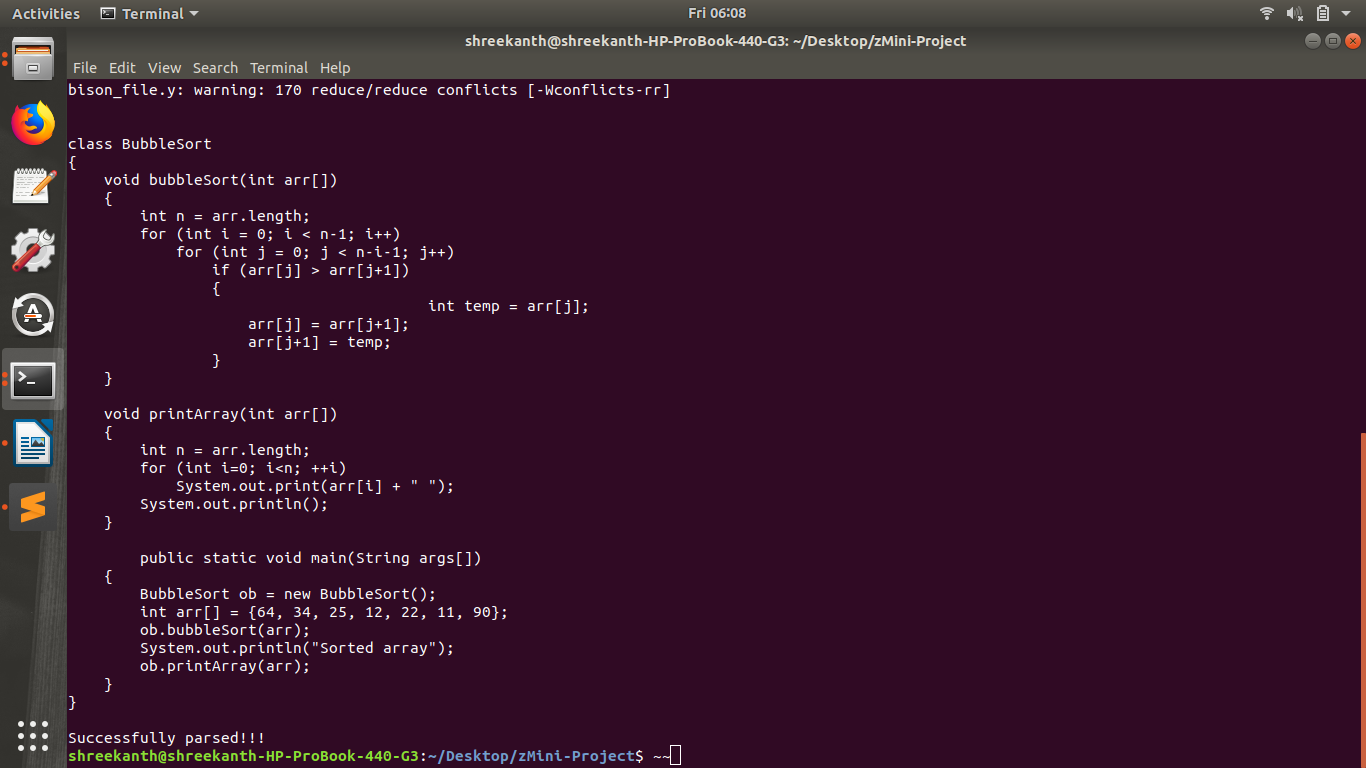
Terminal Output:

**//TEST CASE 2**

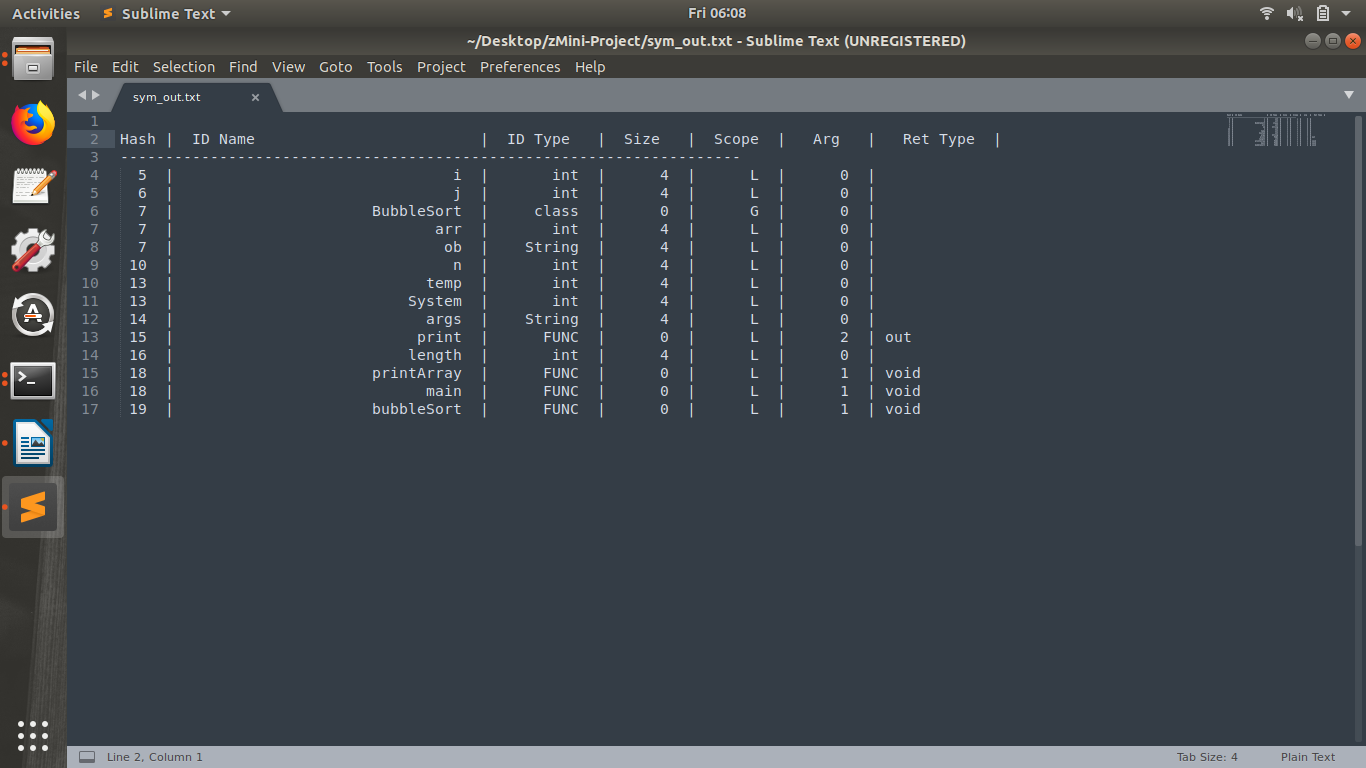
Input:



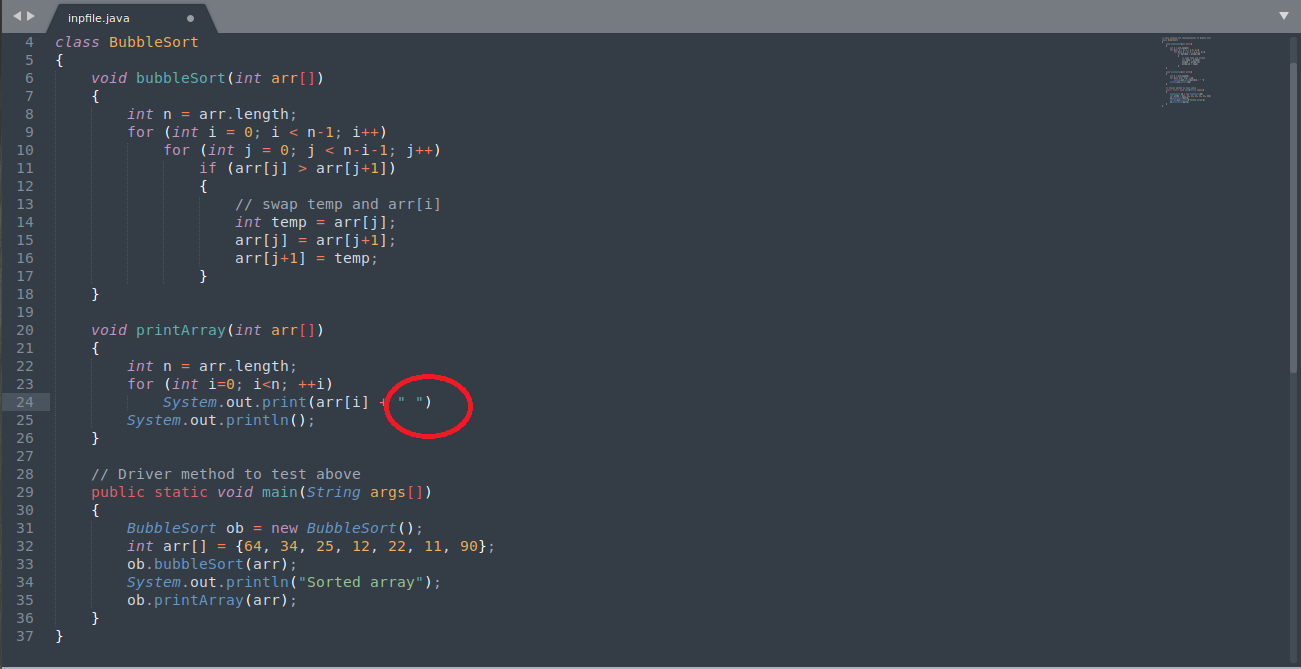
Terminal output:

`

Text File Output



Error Induced Input



Terminal Output:

