1. **INTRODUCTION**

The purpose of this project is to build a simple parser for C language. A parser is a software component that takes input data, frequently in the form of text and gives a structural representation of the input text by building a hierarchical structure, checking for correct syntax in the process. The hierarchical data structure may be a parse tree or an abstract syntax tree. The parser[1] is often preceded by a separate lexical analyzer, which creates tokens from the sequence of input characters.

**1.1 Overview of parser:**

The task of the parser is essentially to determine if and how the input can be derived from the start symbol of the grammar. Parsers may be programmed by hand or may be automatically or semi-automatically generated by a parser generator. Parsers range from very simple functions such as scanf, to complex programs such as the frontend of a C++ compiler.

An important class of simple parsing is done using regular expressions, in which a group of regular expressions defines a regular language and a regular expression engine automatically generating a parser[1] for that language, allowing pattern matching and extraction of text.

Programming languages tend to be specified in terms of a deterministic context-free grammar because fast and efficient parsers can be written for them. For compilers,[2] the parsing itself can be done in one pass or multiple passes.

* 1. **SLR(1) Parser:**

An SLR parser[1] processes the tokens of the input stream by placing them on a stack, and at each point either shifting a token by pushing it onto the stack or reducing some sequence of terminals and non-terminals atop the stack back to some nonterminal symbol.

S stands for simple LR. L means the input string is processed from left to right. 1 stands for one look ahead symbol.

The shift reduce table is a table with rows indexed by states and columns indexed by terminal symbols. When the parser is in some state *s* and the current lookahead terminal is *t*, the action taken by the parser depends on the contents of *action*[*s*][*t*], which can contain four different kinds of entries[3]- Shift, Reduce, Accept and Error.

If the action table entry is shift ‘s’ then push states onto the stack and advance the input so that the look-ahead is set to the next token.If the action table entry is reduce r and rule r has m symbols in its RHS, then pop m symbols off the parse stack. Let s' be the state now revealed on top of the parse stack and N be the LHS non terminal for rule r. Then consult the goto table and push the state given by goto[s'][N] onto the stack. The look ahead token[3] is not changed by this step. If the action table entry is accept, then terminate the parse with success. If the action table entry is error, then signal an error.

The goto table is a table with rows indexed by states and columns indexed by non terminal symbols. When the parser is in state s immediately after reducing by rule N, then the next state to enter is given by goto[s][N].

1. **REQUIREMENT SPECIFICATION**

Software Requirement Specification (SRS) is a complete description of the behavior of the system to be developed. It includes the system requirements which consist of functional and non-functional requirements, user requirements, domain requirements, hardware and software requirements. The functional requirements include what software should do and nonfunctional requirements are related to performance and scalability. This section of SRS contain all the software requirements to a level of detail sufficient to enable designers to design a system to satisfy those requirements and it also help to design their test cases to verify them to check, whether system satisfies those requirements or not.

* 1. **Hardware requirement:**

The hardware requirement is as follows:

* Processor: Intel Core 2 Duo
* In-built RAM: 2.00GB
* System type: LINUX
  1. **Software requirement:**

Following are the minimum software requirements:

* **Compiler:** GNU- GCC compiler
* **Lexical analyzer tool:** LEX

**2.3 Functional Requirement:**

Functional requirements directly support the user requirements by describing the processing of the information as inputs or outputs. Most of the requirements definition focuses mainly on functional requirements, which are based upon the expected functioning of the product system to be created.

* The parser should be able to parse commonly used C language constructs, such as identifiers, expressions, looping and selection statements.
* The parser should detect syntactic errors and display appropriate error message, indicating the type of error and line number where the error is encountered.
* The parser should recover from errors using panic mode recovery and parse the entire program.

**2.4 Non-functional Requirement:**

Non-functional requirements generally support all users. The nonfunctional requirements play a significant part of the specification that is most important for the user. Nonfunctional requirements do not alter the product of functionality.

1. **Reliability**: The parser detects syntax errors correctly and displays appropriate error message.
2. **Ease of use**: The parser displays the complete information about the parsing process to the user like the actions performed and contents of stack at various instances. This helps the user find the exact error in the program and also helps understand the parsing process.
3. **Maintainablity**: The parser is easy to maintain and scale up if required, by adding more constructs.

1. **SYSTEM DESIGN AND DATA STRUCTURES**

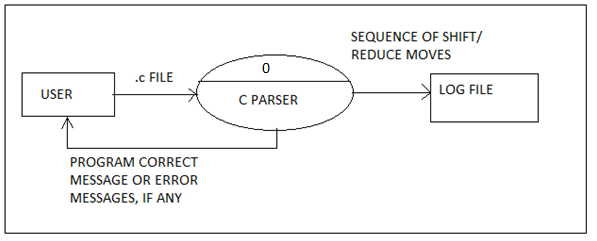
This chapter discuss about the control flow in the software with much more details about software modules by clarifying the details about each function with functionality, purpose, input and output. The decision taken is based on certain design considerations, constraints and dependencies which will affect the subsequent functioning of the product.

**3.1 Data Flow Diagrams:**

A Data Flow Diagram (DFD) is a graphical representation of the flow of data through an information system. Data Flow models are used to show how data flows through a sequence of processing steps. The data transformed at each step before moving on to the next stage. These processing steps or transformations are program functions when Data Flow diagrams are used to document a software design. DFD diagram is composed of four elements, which are process, data flow, external entity and data store.

**3.1.1 Level 0 Data Flow Diagram:**

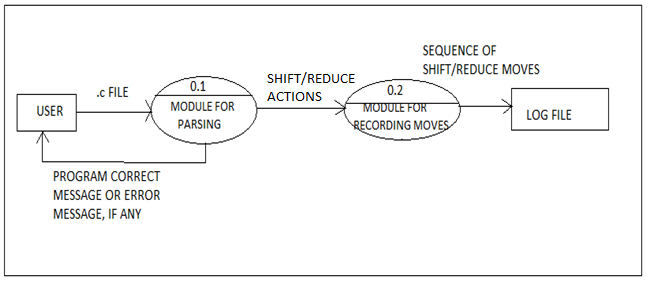
It gives the general flow of data. User gives a .c file as input to the C parser, which then parses the file and displays appropriate error message for any errors encountered. The sequence of actions taken is also recorded in a log file.



**Figure 3.1.1: Level 0 Data Flow Diagram**

* + 1. **Level 1 Data Flow Diagram:**

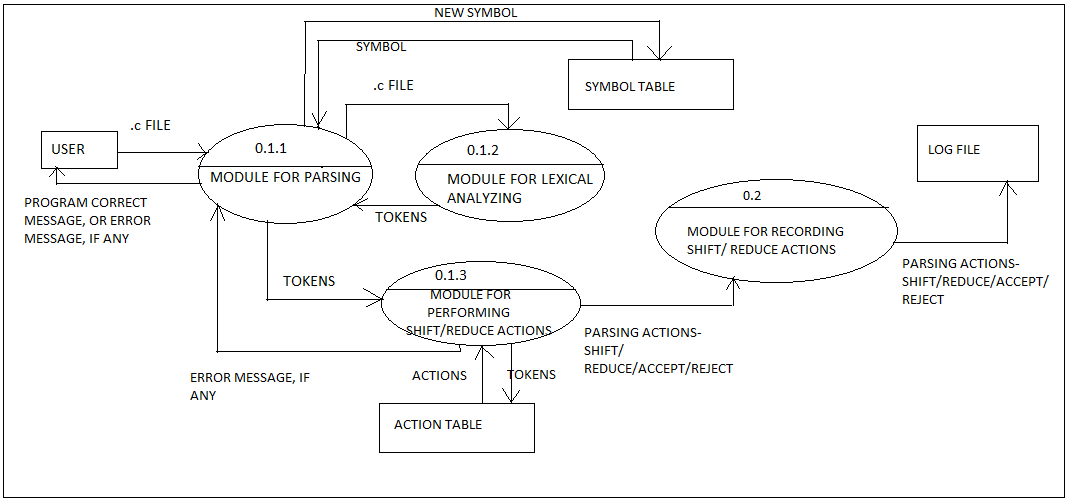
The parser module is divided into two sub-modules- one which carries out parsing actions and other, which records the parsing actions of shift/reduce/accept/reject.

****

**Figure 3.1.2: Level 1 Data Flow Diagram**

* + 1. **Level 2 Data Flow Diagram:**

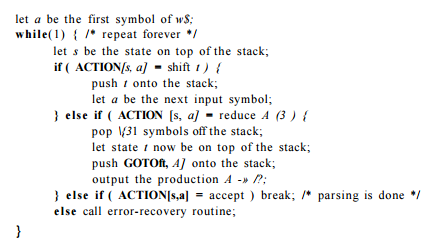
The module for parsing obtains tokens from the module for lexical analysis. It also enters new symbols into the symbol table and obtains symbols from it.



**Figure 3.1.3: Level 2 Data Flow Diagram**

* 1. Algorithm for LR parsing:

The algorithm used is simple LR parsing as shown in the figure below:



**Figure 3.2: LR Parsing algorithm**

* 1. **Data structure used:**

The following data structures are used:

1. **Symbol table:**

A symbol table stores all symbols (identifiers) encountered in the program. A new identifier which is encountered during variable declaration is entered into the symbol table. Any subsequent variable encountered in the program is then verified by checking if it has been entered in the symbol table. If entered, parsing process is continued; otherwise a message of ‘variable not declared’ is displayed. A re declared variable is also detected using the symbol table.

The symbol table has been implemented as a hash table for faster retrieval of symbols, making the parsing process time efficient.

1. **Action table:**

The action table contains the parsing action to be taken for all combinations of symbols in the symbol stack and input token encountered.

If the action table entry is shift ‘s’ then push states onto the stack and advance the input so that the look-ahead is set to the next token.If the action table entry is reduce r and rule r has m symbols in its RHS, then pop m symbols off the parse stack. Let s' be the state now revealed on top of the parse stack and N be the LHS non terminal for rule r. Then consult the goto table and push the state given by goto[s'][N] onto the stack. The look ahead token[3] is not changed by this step. If the action table entry is accept, then terminate the parse with success. If the action table entry is error, then signal an error.

1. **Goto table:**

The goto table is a table with rows indexed by states and columns indexed by non terminal symbols. When the parser is in state s immediately after reducing by rule N, then the next state to enter is given by goto[s][N].

1. **Symbol Stack:**

The symbol stack contains both terminals and non terminals. During shift action, a token is pushed onto the symbol stack. During reduce action, symbols are popped from the stack and the LHS of the appropriate rule is pushed onto it.

1. **State stack:**

The state stack contains information about the state the LR(0) automaton is currently in. State numbers are pushed onto the stack during a shift move and popped during reduce action.

* 1. **Grammar for simple C parser:**

Following are the productions in the simplified C parser grammar:

.

|  |  |
| --- | --- |
| **Production Head** | **Production Body** |
| Program → | int main ( ) Compound-stmt |
| Compound-stmt → | { Local-declarations Stmt-list printf-stmtscanf-stmt } |
| Local-declarations → | Local-declarations Var-declarations | ε |
| Var-declarations → | Type-specifier ID ; | Type-specifier ID [Num] ; |
| Type-specifier → | int | void | double| float |
| Stmt-list → | Stmt-list Stmt | ε |
| Stmt → | Expression-stmt | Compound-stmt | Selection-Stmt | Iteration-stmt | Return-stmt |
| Selection-stmt → | if ( Expression ) Compound-stmt | if ( Expression ) Compound-stmt else Compound-stmt |
| Iteration-stmt → | while ( Expression ) Compound-stmt |  for ( Expression; Expression; Expression) Compond-stmt |
| Return-stmt → | return ; | return Expression ; |
| Expression-stmt → | Var = Expression ; |
| Expression → | Operand Operator Operand | Operand |
| Operand → | Num | Var |
| Var → | ID | ID [ Num ] |
| printf -stmt → | printf(“string list”, Var-list); |
| list → | list specifier string | ε |
| specifier → | %f|%c|%d|%ld |
| Var-list → | Var-list,ID | ε |
| scanf-stmt → | scanf(“Var-list”,&ID); |
| Operator → | RelOp | LogicOp | ArithOp |
| RelOp → | <= | < | >= | > | != | == |
| LogicOp → | && | || |
| ArithOp → | + | - | \* | / | % |

1. **CONCLUSION**

The parser developed can be used to perform syntax analysis for a C program. The program can be directly entered or the filename of the source code can be given as input. The parser generates sequence of shift reduce moves for parsing the given code. The parser is used for validation of variable declarations, arithmetic statements, printf and scanf statements, if else and nested if-else statements, for, while loops.

* 1. **Limitations:**

For simplicity of design the following aspects of C programming language are not implemented.

* Type checking and type conversion
* Pointers and dereferencing
* Function declarations, parameter matching, function call and recursions
* Global and static variables.
* Switch statements, as all switch statements can be converted to if statements.
  1. **Future Enhancement:**
* More extensive grammar taking care of increment, decrement, short hand notation, the ternary operator (?:) and switch statements
* Identifying the number of format specifiers and matching the data types with the variable type in printf and scanf statements.
* Validation of pointer variables, their usage and operations.
* Validating function declarations, parameter matching, function call and recursions
* Declaration of global variables, constant variable and macro variables

1. **REFERENCES**

[1] Alfred W Aho, Monica S Lam, Ravi Sethi, Jeffrey D Ullman, Compilers Principles,

Techniques and Tools, Pearson Education, 2008.

[2] Leland L. Beck, System Software, Third Edtion, Addison-Wesley, 1997.

[3] John R. Levine, Tony Mason and Doug Brown, Lex and Yacc, O’Reilly, SPD, 1999.

[4] HebertSchildt, The Complete Reference C, TMH, Fourth Edition, 2005

**6. APPENDIX-A (SOURCE CODE)**

**6.1 Code for Lex file**

%{

#include<stdio.h>

int count=1;

%}

%%

[\n] lines++;

"for" {return 22;}

"while" {return 21;}

"int" {return 2;}

"float" {return 3;}

[0-9]([0-9]\*) return 6;

"=" return 7;

"+" return 8;

"-" return 9;

"\*" return 10;

"/" return 11;

"(" return 12;

")" return 13;

"if" return 14;

"&&" return 15;

"||" return 16;

"{" return 17;

"}" return 18;

"<"|">"|"<="|">=" return 19;

"=="|"!=" return 20;

[a-zA-Z]([a-z\_A-Z])\* return 1;

[;] return 4;

[$] return 5;

. ;

%%

yywrap()

{

}

**6.2 Code for Parser program**

#include<stdio.h>

#include<string.h>

#include<math.h>

#include<stdlib.h>

void recover();

struct ac

{

charsr;

intsrnum;

}action[100][50];

structrighthand

{

charrprod[100];

};

struct prod

{

intlen;

char lhs[100];

structrighthandrhs[100];

}production[50];

structsym

{

charss[50];

};

intgotor[100][20];

int stacks[100];

structsym symbols[100];

intstackstop=-1;

intsymbolstop=-1;

int k=0;

int ink;

int errors=0;

int lines=0;

int index, index1;

main()

{

char act;

int g;

int length;

inti;

int going;

yyin=fopen("inout.c","r");

stacks[++stackstop]=1;

ink=yylex();

while(1)

{

if(ink==1) index=0;

else if(ink==2) index=1;

else if(ink==3) index=2;

else if(ink==4) index=3;

else if(ink==5) index=4;

else if(ink==6) index=5;

else if(ink==7) index=6;

else if(ink==8) index=7;

else if(ink==9) index=8;

else if(ink==10) index=9;

else if(ink==11) index=10;

else if(ink==12) index=11;

else if(ink==13) index=12;

else if(ink==14) index=13;

else if(ink==15) index=14;

else if(ink==16) index=15;

else if(ink==17) index=16;

else if(ink==18) index=17;

else if(ink==19) index=18;

else if(ink==20) index=19;

else if(ink==21) index=20;

else if(ink==22) index=21;

act=action[stacks[stackstop]][index].sr;

g=action[stacks[stackstop]][index].srnum;

printf("%d, %d\n",stacks[stackstop],index);

if(g==0)

{

//printf(" %d",stacks[stackstop]);

recover(); continue;

}

if(act=='A')

{

if(stackstop==1)

{

printf("accepted %d",stacks[stackstop]) ;

break;

}

else

{recover(); ink=yylex();

continue;

//break;

}

}

else if(act=='S')

{

int l;

symbolstop++;

if(index==0)strcpy(symbols[symbolstop].ss,"IDEN");

if(index==1)strcpy(symbols[symbolstop].ss,"INT");

if(index==2)strcpy(symbols[symbolstop].ss,"FLOAT");

if(index==3)strcpy(symbols[symbolstop].ss,";");

if(index==4)strcpy(symbols[symbolstop].ss,"$");

if(index==5)strcpy(symbols[symbolstop].ss,"CON");

if(index==6)strcpy(symbols[symbolstop].ss,"=");

if(index==7)strcpy(symbols[symbolstop].ss,"+");

if(index==8)strcpy(symbols[symbolstop].ss,"-");

if(index==9)strcpy(symbols[symbolstop].ss,"\*");

if(index==10)strcpy(symbols[symbolstop].ss,"/");

if(index==11)strcpy(symbols[symbolstop].ss,"(");

if(index==12)strcpy(symbols[symbolstop].ss,")");

if(index==13)strcpy(symbols[symbolstop].ss,"IF");

if(index==14)strcpy(symbols[symbolstop].ss,"&&");

if(index==15)strcpy(symbols[symbolstop].ss,"||");

if(index==16)strcpy(symbols[symbolstop].ss,"{");

if(index==17)strcpy(symbols[symbolstop].ss,"}");

if(index==18)strcpy(symbols[symbolstop].ss,"COMP");

if(index==19)strcpy(symbols[symbolstop].ss,"ENQ");

if(index==20)strcpy(symbols[symbolstop].ss,"WHILE");

if(index==21)strcpy(symbols[symbolstop].ss,"FOR");

stacks[++stackstop]=g;

printf("action: %d\n",g);

printf("\n\n");

for(l=0;l<=symbolstop;l++)

printf("symbol %s\n",symbols[l].ss);

int m;

printf("\n\n");

for(m=0;m<=stackstop;m++)

printf("stacks %d\n",stacks[m]);

ink=yylex();

}

else if(act=='R')

{

int l;

int m;

length=production[g].len;

stackstop=stackstop-length;

symbolstop=symbolstop-length;

if(strcmp(production[g].lhs,"S")==0) index1=0;

if(strcmp(production[g].lhs,"decn")==0) index1=1;

if(strcmp(production[g].lhs,"var")==0) index1=2;

if(strcmp(production[g].lhs,"dt")==0) index1=3;

if(strcmp(production[g].lhs,"assn")==0) index1=4;

if(strcmp(production[g].lhs,"exp")==0) index1=5;

if(strcmp(production[g].lhs,"sel")==0) index1=6;

if(strcmp(production[g].lhs,"wloop")==0) index1=7;

if(strcmp(production[g].lhs,"incr")==0) index1=8;

else ;

symbolstop++;

strcpy(symbols[symbolstop].ss,production[g].lhs);

going=gotor[stacks[stackstop]][index1];

stacks[++stackstop]=going;

printf("\n\n");

for(l=0;l<=symbolstop;l++)

printf("symbol %s\n",symbols[l].ss);

printf("\n\n");

for(m=0;m<=stackstop;m++)

printf("stacks %d\n",stacks[m]);

}

if(stackstop==-1 || symbolstop==-1)break;

}

if(act=='A' &&stackstop==1) {if(errors==0)printf("\nsyntactically correct\n");}

}

void recover()

{ intl,m; ++errors; char a[10];

printf("rejected\n");

if(strcmp(symbols[symbolstop].ss,"S")==0 || stacks[stackstop]==49)

{printf("error on line %d: missing }\n", lines);

strcpy(symbols[++symbolstop].ss,"}");

stacks[++stackstop]=51;

}

else if(stacks[stackstop]==28)

{printf("error on line %d: missing ) in expression\n",lines);

strcpy(symbols[++symbolstop].ss,")");

stacks[++stackstop]=29;

}

else if(strcmp(symbols[symbolstop].ss,"=")==0)

{printf("error on line %d: missing expression\n",lines);

strcpy(symbols[++symbolstop].ss,"IDEN");

stacks[++stackstop]=10;

}

else if(stacks[stackstop]==19 || stacks[stackstop]==20 || stacks[stackstop]==21 ||stacks[stackstop]==22 || stacks[stackstop]==37 )

{printf("error on line %d: missing operand\n",lines);

strcpy(symbols[++symbolstop].ss,"IDEN");

stacks[++stackstop]=10;

}

else if(strcmp(symbols[symbolstop].ss,"}")==0 || stacks[stackstop]==51)

{printf("error on line %d: extra }\n", lines);

exit(0);

}

/\* else if(stacks[stackstop]==28)

{printf("error on line %d: missing ) in expression\n",lines);

strcpy(symbols[++symbolstop].ss,")");

//stacks[++stackstop]=29;

}\*/

/\*else if(stacks[stackstop]==7)

{printf("error on line %d: missing ; in expression\n",lines);

strcpy(symbols[++symbolstop].ss,";"); ink=yylex();

//stacks[++stackstop]=29;

}\*/

else if(stacks[stackstop]==10)

{printf("error on line %d: missing ; in expression\n",lines);

while(stacks[stackstop]!=49){stackstop--;symbolstop--;}

//stacks[++stackstop]=29;

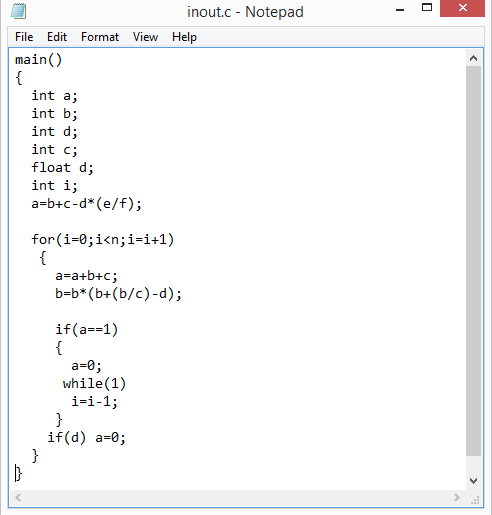
}

else exit(0);

}

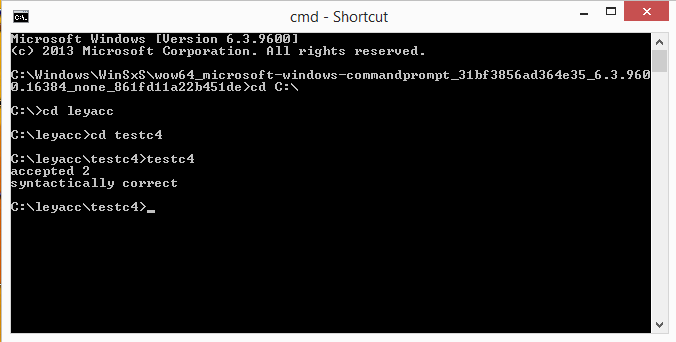
**7. APPENDIX B (SCREENSHOTS)**

**7.1 Input program 1:**



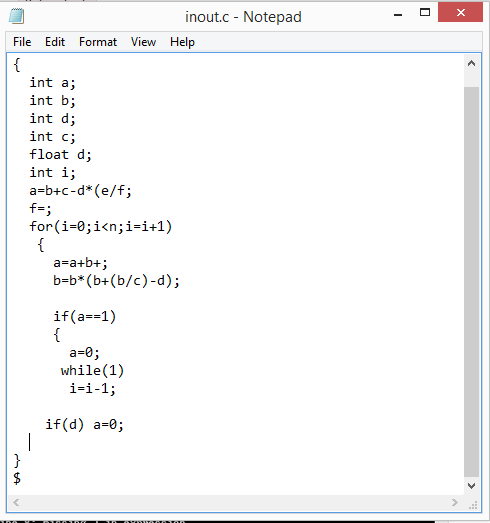
**Figure A1: Input program 1**

**7.2 Output for program 1:**

****

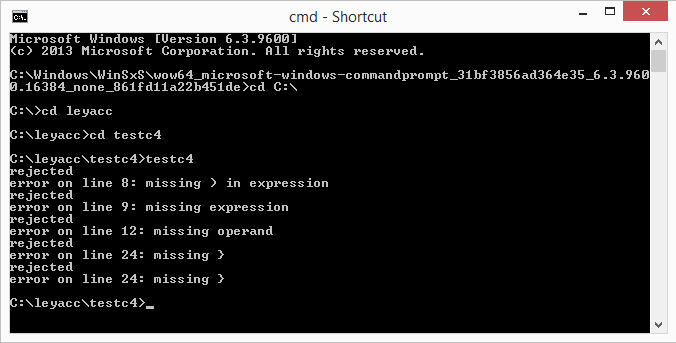
**Figure A2: Output for program 1**

**7.3 Input program 2 with unbalanced parenthesis:**

****

**Figure A3: Input program 2 with unbalanced parenthesis**

**7.4 Error message for input program 2**

****

**Figure A4: Error message for input program 2**