Lexical Syntax and Semantic Analyzer

Documentation

Asad Ali

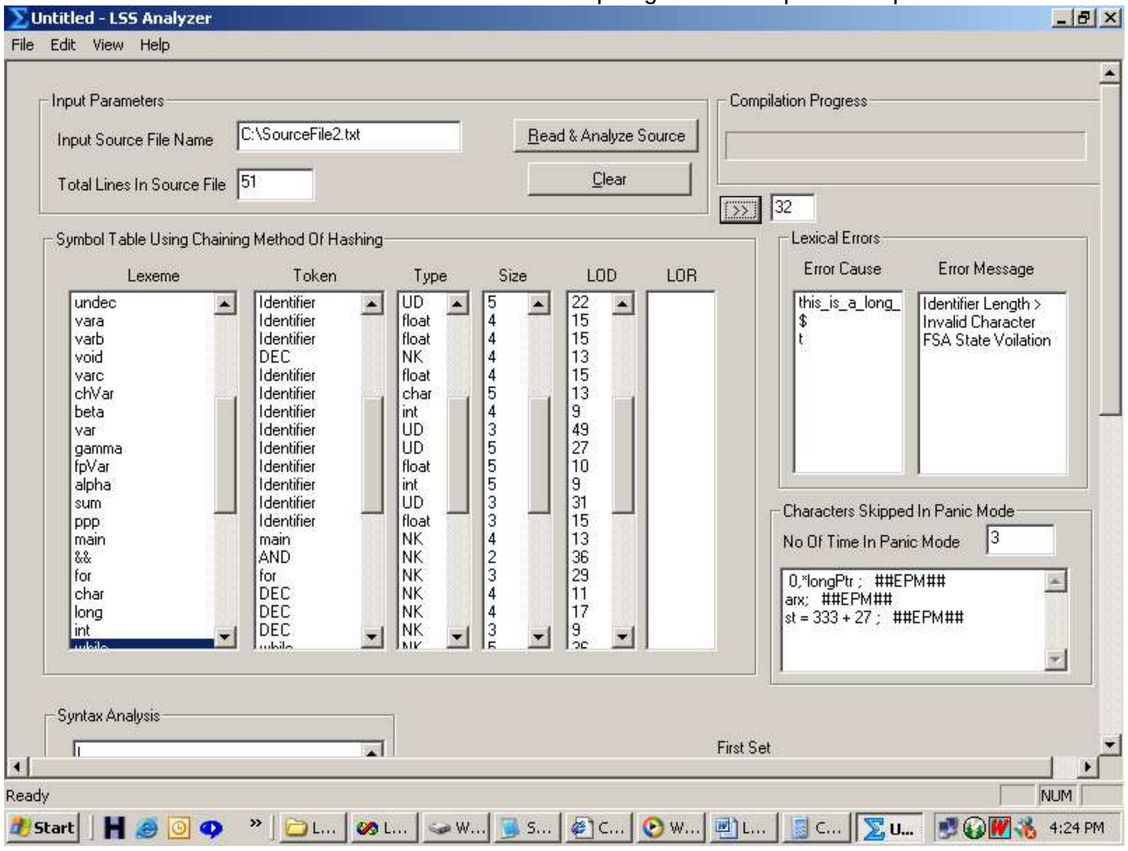
The documentation below describes the constructs that have been implemented using the standard C language and our implementation methodology.

1. **Lexical Analysis:** In the lexical analysis phase Finite state automata have been implemented for the following C constructs: **Pre-processor, declaration statements, functions, assignment statement, for statement, while statement.**
2. Individual state machines have been implemented hence besides returning the token of the related lexeme the automata also verify the syntax of the above mentioned constructs.
3. Lexical analyzer maintains a symbol table for all the lexemes present in the source file. The symbol table has been implemented using the Chaining method of hashing.
4. In Chaining method of hashing separate link lists are maintained after the initial hash index is generated using the lexeme.
5. Other information like identifier type, line of declaration, line of reference are also maintained by the symbol table.
6. Error recovery techniques have also been implemented in the lexical phase using the panic mode.
7. In panic mode when ever a lexical error occurs like unknown character, variable length greater than valid identifier limit, are encountered, the compiler enters a special state in which it start skipping characters until a synchronizing token is encountered which in our case is a semicolon.
8. Brackets in the source program are handled by a special automata which uses a stack push pop mechanism for matching the braces.
9. In order to maintain information about the symbol table and line of references two structures named Symbol table and Line of Reference were used.
10. When ever a Finite state automata of a particular construct is violated a FSA violation error is generated which indicated in-fact the syntax violation.
11. **Syntax Analysis:** Syntax analysis of the assignment statement has been done using the first set of its grammar. Also the syntax of the all the above mentioned constructs are verified using the Finite state automata of the respective constructs.
12. First of all left recursive grammar was converted to right recursive and then the first set identifying grammar statements for all the terminal symbols was used to construct a table.
13. This table known as the parsing table or first set was then used by the non-recursive Predictive parser to derive the assignment statement from its grammar.
14. This in effect verifies the syntax of the assignment statement.
15. **Semantic Analyses:** Semantics of the assignment statement have been verified also. Semantics analysis only checks for the static semantics like type compatibility and variable being un-declared.

16. For this purpose the lexemes in an assignment statement are cached until the syntax of the assignment statement is verified to be correct.
17. After which types of each one of them is queried from the symbol table.
18. Then first the type of the right hand side of the assignment statement is computed by comparing the adjacent types of the variables and checking for the validity of the operation defined on them.
19. After the right hand side type is determined, its is again compared for assignment compatibility with the left hand side of the assignment statement.
20. Below is the C source code used in the analysis:

```
#include <iostream.h>
#include <conio.h>

// this is a single line comment
/* this is a
multi line comment * fsa test */

int alpha = 0,beta=10;
float fpVar = 0;
char chVar;

void main()
{ // this is a comment and should be skipped.
float varx,vara,varb,varc,ppp,test ; // again comment
int varint ;
long this_is_a_long_Type_Variable = 0,*longPtr ;

varx = 10 * varc + vara; // place them above later
varx = varb + varc ;
chVar = 10 * varc;
fpVar = undec+beta;

varx = varb + varc ;
alpha = 10;
alpha = beta + 10 * $varx;
gamma = 56;

for(i=0,j=beta; i<10,j>=20;i+=1,j-=2)
{
   sum = alpha * i;
   sum=sum+j;
}


while(varx !=vara && varb <=varc)
{
   varx=varx+1;
   varb = varx * 30;
}
/* Another Multiline Comment
to test the program */
vara = varb - varc
test = 333 + 27 ;
/* Compute addition & Subtraction of Variables
and Display the Results */
varint = vara + 90 ;
varint = var *  21 ;
}
```

Shown below is the screen shot of the programs sample output.



Input Source File Name: C:\SourceFile2.txt

Total Lines In Source File: 51

32

Symbol Table Using Chaining Method Of Hashing

Lexeme	Token	Type	Size	LOD	LOR
undec	Identifier	UD	5	22	
vara	Identifier	float	4	15	
varb	Identifier	float	4	15	
void	DEC	NK	4	13	
varc	Identifier	float	4	15	
chVar	Identifier	char	5	13	
beta	Identifier	int	4	9	
var	Identifier	UD	3	49	
gamma	Identifier	UD	5	27	
fpVar	Identifier	float	5	10	
alpha	Identifier	int	5	9	
sum	Identifier	UD	3	31	
ppp	Identifier	float	3	15	
main	main	NK	4	13	
&&	AND	NK	2	36	
for	for	NK	3	29	
char	DEC	NK	4	11	
long	DEC	NK	4	17	
int	DEC	NK	3	9	

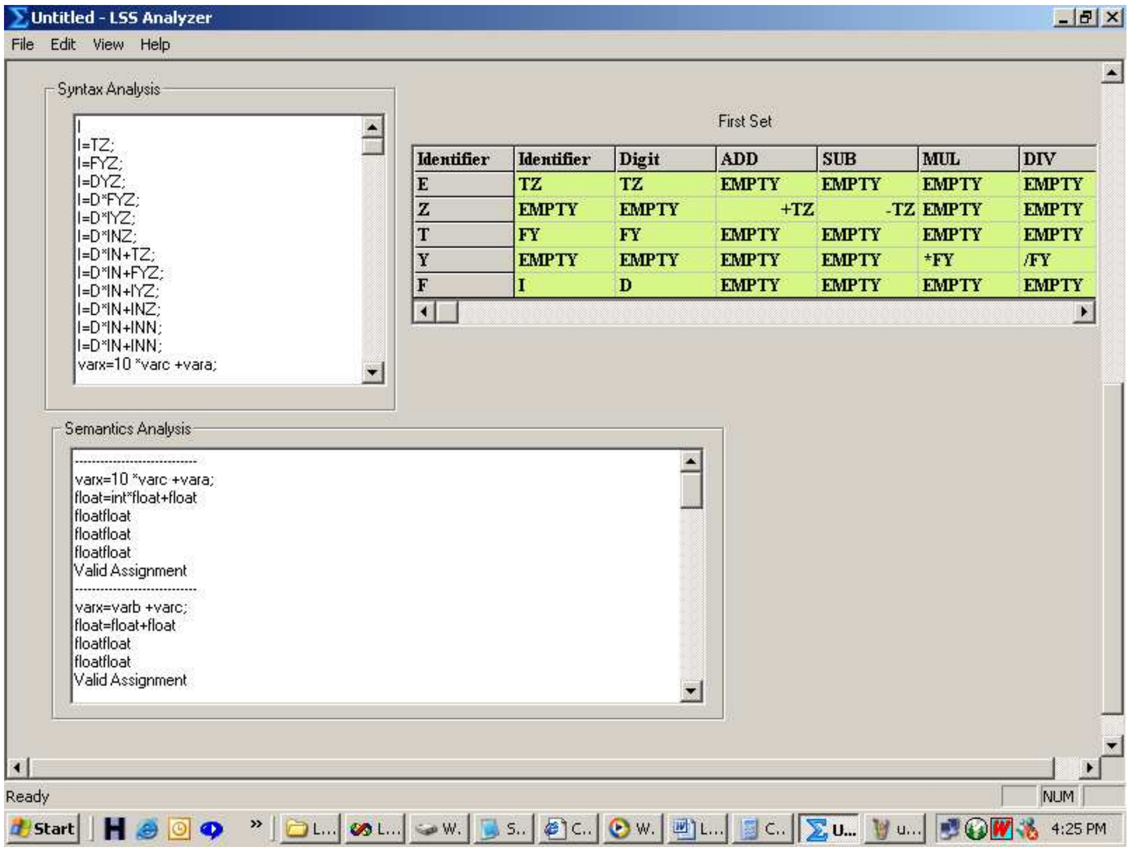
Lexical Errors

Error Cause	Error Message
this_is_a_long_	Identifier Length >
$	Invalid Character
t	FSA State Voilation

Characters Skipped In Panic Mode

No Of Time In Panic Mode: 3

```
 0,*longPtr ;   ##EPM##
arx;   ##EPM##
st = 333 + 27 ;   ##EPM##
```



Untitled - LSS Analyzer

Syntax Analysis

```
I
I=TZ;
I=FYZ;
I=DYZ;
I=D*FYZ;
I=D*IYZ;
I=D*INZ;
I=D*IN+TZ;
I=D*IN+FYZ;
I=D*IN+IYZ;
I=D*IN+INZ;
I=D*IN+INN;
I=D*IN+INN;
varx=10 *varc +vara;
```

First Set

Identifier	Identifier	Digit	ADD	SUB	MUL	DIV
E	TZ	TZ	EMPTY	EMPTY	EMPTY	EMPTY
Z	EMPTY	EMPTY	+TZ	-TZ	EMPTY	EMPTY
T	FY	FY	EMPTY	EMPTY	EMPTY	EMPTY
Y	EMPTY	EMPTY	EMPTY	EMPTY	*FY	/FY
F	I	D	EMPTY	EMPTY	EMPTY	EMPTY

Semantics Analysis

```
----------------------------
varx=10 *varc +vara;
float=int*float+float
floatfloat
floatfloat
floatfloat
Valid Assignment
----------------------------
varx=varb +varc;
float=float+float
floatfloat
floatfloat
Valid Assignment
```

Above is the screen shot of the lower half of the program window hidden previously.