Session 3: Symbol Table Construction and Management

I. OBJECTIVES:

The main purpose of this session is to introduce the symbol table, the table in which all the identifiers are stored along with information about them. When a variable is declared, the compiler enters it as a new entry in the symbol table. When a variable is referred to in an expression, the compiler looks up in the symbol table to retrieve necessary information about it, such as its data type, value, etc., and the compiler performs other actions on the table like delete, update and so on.

II. DEMONSTRATION OF USEFUL RESOURCES:

Related sample programs will be demonstrated.

III. LAB EXERCISE:

Sept 1 and Step 2 of the Assignment #3 described below.

IV. ASSIGNMENT #3:

Suppose, a given C source program has been scanned, filtered and then lexically analyzed as it was done in Session 1 & 2. We have all the lexemes marked as different types of tokens like keywords, identifiers, operators, separators, parentheses, numbers, etc. Now we generate a Symbol Table describing the features of the identifiers. Then, we generate a modified token stream in accordance with the Symbol Table for processing by the next phase, that is, Syntax Analysis.

Sample source program:

Sample source program:

```
// A program fragment
float x1 = 3.125;
/* Definition of the
function f1 */
double  f1(int x)
{
  double z;
  z = 0.01;
  return z;
}
//* Beginning of 'main'
int main(void)
{
 int n1;  double z;
 n1=25;  z=f1(n1);
```

Sample input based on the program fragment:

[kw float] [id x1] [op =] [num 3.125] [sep ;] [kw double] [id f1] [par (] [kw int] [id x] [par)] [brc {] [kw double] [id z] [sep ;] [id z] [op =] [num 0.01] [sep ;] [kw return] [id z] [sep ;] [brc }] [kw int] [id main] [par (] [kw void] [par)] [brc {] [kw int] [id n1] [sep ;] [kw double] [id z] [sep ;] [id n1] [op =] [num 25] [sep ;] [id z] [op =] [id f1] [par (] [id n1] [par)] [sep ;]

Sample input based on the program fragment:

[kw float] [id x1] [op =] [num 3.125] [sep ;] [kw double] [id f1] [par (] [kw int] [id x] [par)] [brc {] [kw double] [id z] [sep ;] [id z] [op =] [num 0.01] [op +] [id x] [op *] [num 5.5] [sep ;] [kw return] [id z] [sep ;] [brc }] [kwint] [id main] [par (] [kw void] [par)] [brc {] [kw int] [id n1] [sep ;] [kw double] [id z] [sep ;] [id n1] [op =] [num 25] [sep ;] [id z] [op =] [id f1] [par (] [id n1] [par)] [sep ;]

Step 1: After complete recognition of all the lexemes only identifiers are kept in pairs for formation of Symbol Tables. The token stream should look like the one as follows:

[float] [id x1] [=] [3.125] [;] [double] [id f1] [(] [int] [id x] [)] [{] [double] [id z] [;] [id z] [=] [0.01] [;] [return] [id z] [;] **[}]** [int] [id main] [(] [void] [)] [{] [int] [id n1] [;] [double] [id z] [;] [id n1] [=] [25] [;] [id z] [=] [id f1] [(] [id n1] [)] [;]

Step 2: Symbol Table generation:

Sample source program:

```
// A program fragment
float x1 = 3.125;
/* Definition of the
function f1 */
double  f1(int x)
{
  double z;
  z = 0.01;
  return z;
}
//* Beginning of 'main'
int main(void)
{
 int n1;  double z;
 n1=25;  z=f1(n1);
```

Symbol Table:

Sl. No.	*Name*	*Id Type*	*Data Type*	*Scope*	*Value*
1	x1	var	float	global	3.125
2	f1	func	double	global	
3	x	var	int	f1	
4	z	var	double	f1	0.01
5	main	func	int	global	
6	n1	var	int	main	25
7	z	var	double	main	

<u>Step 3:</u> Your program should implement the following functions on symbol table.

1. *insert()*
2. *update()*
3. *delete()*
4. *search()*
5. *display()*

<u>Step 4:</u> Modified token stream for Syntax Analysis:

<u>Sample source program:</u>

```
// A program fragment
float x1 = 3.125;
/* Definition of the
function f1 */
double  f1(int x)
{
  double z;
  z = 0.01;
  return z;
}
//* Beginning of 'main'
int main(void)
{
int n1;  double z;
 n1=25;  z=f1(n1);
```

[float] [id 1] [=] [3.125] [;] [double] [id 2] [(] [int] [id 3] [)] [{] [double] [id 4] [;] [id 4] [=] [0.01] [;] [return] [id 4] [;] [}] [int] [id 5] [(] [void] [)] [{] [int] [id 6] [;] [double] [id 7] [;] [id 6] [=] [25] [;] [id 7] [=] [id 2] [(] [id 6] [)] [;]

Session 4: Detecting Simple Syntax Errors

I. OBJECTIVE:

Syntax errors are very common in source programs. The main purpose of this session is to write programs to detect and report simple syntax errors.

II. DEMONSTRATION OF USEFUL RESOURCES:

Sample programs will be demonstrated related to syntax error detection.

III. LAB EXERCISE:

Write programs to detect the following syntax errors.

1. Duplicate Identifier Declarations.
2. Unbalanced curly braces Detection.

IV. ASSIGNMENT #4:

Suppose, a given C source program has been scanned, filtered, lexically analyzed and tokenized as that were done in earlier sessions. In addition, line numbers have been assigned to the source code lines for generating proper error messages. As the first step to Syntax Analysis, we now perform detection of simple syntax errors like duplication of tokens except parentheses or braces, unbalanced braces or parentheses problem, unmatched 'else' problem, etc. Duplicate identifier declarations must also be detected with the help of the Symbol Table.

<u>Sample Input:</u> Sample code segment with numerous syntax errors.

```
/* A program fragment*/

float     x1 = 3.125;;;
/* Definition of function f1 */
double  f1(float a, int int x)
{if(x<x1)
double z;;
else z =          0.01;}}
else   return z;
}
/* Beginning of 'main'      */
int main(void)
{{{{
int n1;  double z;
n1=25;  z=f1(n1);}
```

<u>Intermediate Output:</u> Recognized tokens in the lines of code.

```
1
2
3 kw float id x1 = 3.125 ; ; ;
4
5 double id f1 ( float id a , int int id x )
6 { if ( id x < id x1 )
7 double id z ; ;
8 else id z = 0.01 ; } }
9 else return id z ;
10 }
11
12 int id main ( void )
13 { { { {
14 int id n1 ; double id z ;
15 id n1 = 25 ; id z = id f1 ( id n1 ) ; }
16
```

Sample Output: Types of detected errors

Duplicate token at line 3, Misplaced ‘}’ at line 8, Unmatched ‘else’ at line 9, etc.

Guidelines:

1. Unbalanced braces or parentheses problem in an arithmetic or relational expression can be detected during tokenization in a simple way by counting the openings and closings. Stack can be used here as well.
2. Unmatched ‘else’ problem in its simplest form may also be detected by counting ‘if’s and ‘else’s: For every ‘else’ there must be an ‘if’ that occurs earlier.
3. Undeclared identifiers and duplicate identifier declarations in the same scope are detected during Symbol Table construction in a relatively easier way.
4. Duplicate ‘;’ in ‘for’ construct of C demands additional checking. *for(;;){}*