Assignment no : 07

Write a program using LEX and YACC to generate a symbol table

Assign7.c

```
#include <iostream>
#include <cctype>
#include <cstdlib>
#include <cstring>

using namespace std;

int main() {
    int i = 0, x = 0, n;
    void *p, *add[5];
    char b[50], d[15], c;

    cout << "Expression terminated by $:";
    while ((c = getchar()) != '$') {
        b[i] = c;
        i++;
    }
    n = i - 1;

    cout << "Given Expression:";
    for (i = 0; i <= n; i++) {
        cout << b[i];
    }

    cout << "\nSymbol Table\n";
    cout << "Symbol \t addr \t size \t type\n";

    for (int j = 0; j <= n; j++) {
        c = b[j];
        if (isalpha(c)) {
            int size = sizeof(char); // Size of character is 1 byte
            p = malloc(size);
            add[x] = p;
            d[x] = c;
            cout << c << " \t " << p << " \t " << size << " \t identifier\n";
            x++;
        } else if (c == '+' || c == '-' || c == '*' || c == '=') {
```

```
            int size = sizeof(char); // Size of character is 1 byte
            p = malloc(size);
            add[x] = p;
            d[x] = c;
            cout << c << " \t " << p << " \t " << size << " \t operator\n";
            x++;
        }
    }

    return 0;
}
```

Output:

gcc assign7.c

./a.out

Explanation :

This C++ program is designed to generate a symbol table for a given expression. Let's break down the functionality and logic step by step:

1. **Input**: The program expects an expression terminated by '$'. It reads characters until it encounters '$' using `getchar()`.

2. **Storage**: It stores the input expression character by character in an array named `b`.

3. **Symbol Table Generation**:

- It iterates through each character in the input expression.

- If the character is an alphabet (`isalpha(c)`), it considers it as an identifier and allocates memory for it using `malloc()`. The address of the allocated memory is stored in the array `add[]`, and the character itself is stored in the array `d[]`. It prints the symbol, its address, size (1 byte for a character), and type (identifier).

- If the character is one of the specified operators ('+', '-', '*', '='), it treats it as an operator. Similar to identifiers, it allocates memory for it, stores its address in `add[]`, and stores the operator itself in `d[]`. It then prints the symbol, its address, size (1 byte for a character), and type (operator).

4. **Output**: Finally, it prints the generated symbol table with columns for symbol, address, size, and type.

However, this code has a few issues:

- It uses `malloc()` without `free()`, leading to memory leaks.

- It doesn't handle other types of symbols such as digits, parentheses, etc.

- It's limited to single-character symbols.

To improve the code, you might want to consider using dynamic memory allocation properly, handling multi-character symbols, and extending support for different types of symbols. Additionally, using lexical analyzer (LEX) and parser (YACC) could provide a more robust and scalable solution for symbol table generation, especially for handling complex expressions and languages.

Certainly! Let's delve into some relevant theory behind symbol tables, lexical analysis, and parsing:

1. **Symbol Table**:

   - In programming language theory, a symbol table is a data structure used by a compiler or interpreter to store information about the variables, functions, classes, and other symbols used in a program.

   - It typically includes entries for each symbol encountered during the compilation or interpretation process, along with information such as its name, type, scope, memory address, and other attributes.

   - Symbol tables are essential for various compiler and interpreter tasks, including semantic analysis, code generation, optimization, and error checking.

2. **Lexical Analysis**:

   - Lexical analysis, also known as scanning or tokenization, is the first phase of a compiler or interpreter.

   - Its primary task is to read the input source code character by character and group them into meaningful units called tokens.

   - Tokens represent the smallest meaningful units in a programming language, such as keywords, identifiers, literals, operators, and punctuation symbols.

- Lexical analyzers typically use finite automata, regular expressions, or other techniques to recognize and classify tokens efficiently.

3. **Parser**:

- Parsing is the process of analyzing a sequence of tokens (generated by the lexical analyzer) to determine its syntactic structure according to the rules of a formal grammar.

- A parser takes the sequence of tokens as input and constructs a parse tree or abstract syntax tree (AST) that represents the hierarchical structure of the input code.

- There are different types of parsers, including top-down parsers (e.g., LL parsers) and bottom-up parsers (e.g., LR parsers), each following different parsing techniques and algorithms.

- Parsers play a crucial role in compiler construction, as they enable the compiler to understand the syntactic structure of the source code, perform semantic analysis, and generate intermediate representations or executable code.

4. **LEX and YACC**:

- LEX and YACC are tools commonly used in compiler construction to generate lexical analyzers (scanners) and parsers, respectively.

- LEX is a lexical analyzer generator that takes regular expressions and associated actions as input and generates efficient C code for tokenization.

- YACC (Yet Another Compiler Compiler) is a parser generator that takes a formal grammar specification in BNF (Backus-Naur Form) notation and generates C code for a parser based on the specified grammar rules.

- By combining LEX and YACC, developers can build complete compilers or interpreters for programming languages efficiently, as these tools automate the generation of key components such as lexical analyzers and parsers.

In summary, symbol tables, lexical analysis, parsing, and tools like LEX and YACC are fundamental concepts and tools in compiler construction and programming language implementation, enabling the transformation of source code into executable programs.