Abstract

This project aims to implement a simple arithmetic expression evaluator using Syntax-Directed Translation (SDT). It leverages Lex for lexical analysis and Yacc for syntax analysis and evaluation. The calculator accepts expressions involving addition, subtraction, multiplication, division, and parentheses. It processes expressions based on context-free grammar and evaluates them using the semantic rules attached to grammar productions. This project demonstrates key compiler phases such as lexical analysis, parsing, syntax tree construction, and immediate expression evaluation. It serves as a hands-on application of theoretical compiler design concepts in a practical and interactive format

Introduction

Compilers translate high-level programming code into machine-readable instructions through several stages, including lexical analysis, parsing, semantic analysis, and code generation. In this mini-project, we focus on the first three stages by building a calculator using Lex and Yacc tools.

Lex identifies the tokens in the input, such as numbers and operators, while Yacc parses these tokens according to a predefined grammar and evaluates the expressions using syntax-directed translation techniques. This allows the calculator to handle operator precedence, associativity, and nested

Existing System

In the existing systems, basic calculators are implemented using traditional programming constructs such as loops, conditionals, and functions without following compiler design principles. These calculators often:

- Do not parse expressions using grammar.
- Evaluate expressions linearly without proper operator precedence.
- Lack flexibility to handle nested parentheses or complex expressions.
- Do not demonstrate compiler phases like lexical and syntax analysis.
- Are not extendable for future compiler functionalities.

As a result, such implementations are limited in scope and fail to model how real compilers process source code.

Proposed System

The proposed system is a grammar-driven calculator that uses Syntax-Directed Translation with Lex and Yacc to evaluate arithmetic expressions. This system:

- Implements a formal grammar to parse expressions.
- Uses Lex for tokenizing input into meaningful components (numbers, operators).
- Uses Yacc to parse input based on grammar rules and perform evaluation.
- Supports operator precedence and associativity usir grammar rules.
- Can handle parenthesized expressions and nested operations.
- Provides a solid foundation for learning and implementing real compiler phases.
- **Advantages**:

- Models real compiler behaviour.
- Easy to extend for variables, functions, etc.
- Better error handling and structure.

Hardware Requirements

Component.	**Specification**
Processor.	Intel Pentium IV or a /AMD equivalent
RAM.	Minimum 2 GB (4 GB recommended)
Storage	Minimum 100 MB of free disk space
Input Device.	Keyboard
Output Device.	Monitor
Operating System.	Linux (preferred) / Windows with tools installed

Software Requirements

Software/Tool.	Purpose
Lex (Flex).	Lexical Analyzer Generator
Yacc (Bison).	Syntax Parser Generator (for SDT)
GCC Compiler.	Compilation of C files (Lex/Yacc output)
Terminal / Shell.	Command-line interface for execution
Text Editor.	Code editing (e.g., Vim, Nano, VS Code, gedit)
Operating System.	Linux (Ubuntu/Debian/Fedora), or Windows with WSL or ↓ W

Implementation

- **Files Required**

1.Calc.l (Lex file)

```
%{
#include "y.tab.h"
%}

Digit   [0-9]+

%%

{digit} { yylval = atoi(yytext); return NUM; }
[+\-*/()] { return yytext[0]; }
[ \t\n]  ;  // Skip whitespaces
.     { printf("Invalid character: %s\n", yytext); }

%
```

2.calc.y (Yacc file)

```
%{
#include <stdio.h>
#include <stdlib.h>
%}

%token NUM
%left '+' '-'
%left '*' '/'
```

```
%start input

%%

Input:
  Input expr '\n'  { printf("= %d\n", $2); }
 | /* empty */
 ;

Expr:
  Expr '+' expr    { $$ = $1 + $3; }
 | expr '-' expr    { $$ = $1 - $3; }
 | expr '*' expr    { $$ = $1 * $3; }
 | expr '/' expr    {
    If ($3 == 0) {
      Printf("Error: Division by zero\n");
      Exit(1);
    }
    $$ = $1 / $3;
  }
 | '(' expr ')'    { $$ = $2; }
 | NUM             { $$ = $1; }
 ;

%%
```

```
Int main() {
  Printf("Enter expressions (Ctrl+D to quit):\n");
  Yyparse();
  Return 0;
}

Int yyerror(char *s) {
  Fprintf(stderr, "Syntax Error: %s\n", s);
  Return 1;
}
```

Steps to Compile and Run

Yacc -d calc.y

Lex calc.l

Gcc y.tab.c lex.yy.c -o calc -ll

./calc

Sample Output

- **Valid inputs**
  Enter expressions (Ctrl+D to quit):

5 + 2 * 3

= 11

(10 + 4) / 2

= 7

- **Division by zero**
  5 / 0
  Error: Division by zero
- **Syntax Error**
  3 + * 2
  Syntax Error: syntax error

Conclusion

This project demonstrates how compiler techniques like lexical analysis, parsing, and syntax-directed translation can be applied to a real problem: expression evaluation. By using Lex and Yacc, the project gives students hands-on experience with parsing, grammar design, operator precedence, and semantic actions, building a strong foundation for compiler construction.