# Experiment 2

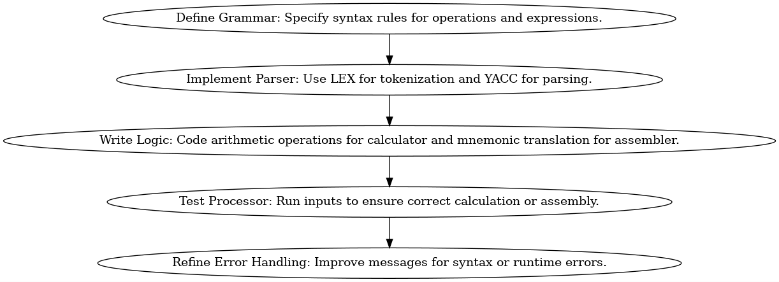
## Aim

Develop simple language processors like desk calculator

**Steps:**

1. Define the grammar of the language, specifying the syntax for operations and expressions for the calculator, and the mnemonic to opcode translations for the assembler.
2. Implement a parser using a tool like LEX for tokenizing input and YACC for parsing according to the defined grammar, to interpret or compile the language.
3. Write the evaluation logic for the calculator to perform arithmetic operations and store variables, or for the assembler, translate mnemonics to machine code.
4. Test the processor with various inputs to ensure it correctly calculates expressions or assembles code into executable machine instructions.
5. Refine error handling to provide meaningful messages for syntax or runtime errors encountered by users.

**Flowchart:**



## Code

|  |
| --- |
| #include <stdio.h> #include <math.h>  void **print\_menu**() {  printf("\n--- Calculator Menu ---\n");  printf("1: Addition (+)\n");  printf("2: Subtraction (-)\n");  printf("3: Multiplication (\*)\n");  printf("4: Division (/)\n");  printf("5: Modulus (mod)\n");  printf("6: Exponentiation (pow)\n");  printf("7: Sine (sin)\n");  printf("8: Cosine (cos)\n");  printf("9: Tangent (tan)\n");  printf("0: Exit\n");  printf("Enter your choice: "); }  double **get\_number**() {  double num;  scanf("%lf", &num);  return num; }  int **main**() {  int operation;  double num1, num2, result;   while (1) {  print\_menu();  scanf("%d", &operation);   if (operation == 0) {  break;  }   switch (operation) {  case 1:  printf("Enter two numbers: ");  num1 = get\_number();  num2 = get\_number();  result = num1 + num2;  break;  case 2:  printf("Enter two numbers: ");  num1 = get\_number();  num2 = get\_number();  result = num1 - num2;  break;  case 3:  printf("Enter two numbers: ");  num1 = get\_number();  num2 = get\_number();  result = num1 \* num2;  break;  case 4:  printf("Enter two numbers: ");  num1 = get\_number();  num2 = get\_number();  if (num2 != 0) {  result = num1 / num2;  } else {  printf("Error: Division by zero!\n");  continue;  }  break;  case 5:  printf("Enter two numbers: ");  num1 = get\_number();  num2 = get\_number();  result = fmod(num1, num2);  break;  case 6:  printf("Enter base and exponent: ");  num1 = get\_number();  num2 = get\_number();  result = pow(num1, num2);  break;  case 7:  printf("Enter angle in radians: ");  num1 = get\_number();  result = sin(num1);  break;  case 8:  printf("Enter angle in radians: ");  num1 = get\_number();  result = cos(num1);  break;  case 9:  printf("Enter angle in radians: ");  num1 = get\_number();  result = tan(num1);  break;  default:  printf("Invalid operation!\n");  continue;  }   printf("Result: %.2lf\n", result);  }   printf("Calculator terminated. Goodbye!\n");  return 0; } |

## Output

# 

# 

# 

# 

# 

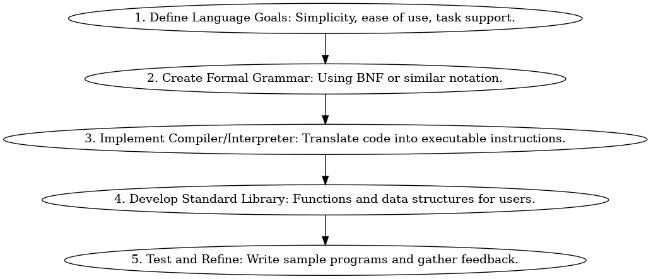
# Experiment 2 x

Design a small high-level language.

**Steps:**

1. Define the language's goals and features, such as simplicity, ease of use, and specific tasks it should support.
2. Create a formal grammar for the language, specifying its syntax and rules using a Backus-Naur Form (BNF) or similar notation.
3. Implement a compiler or interpreter that translates the language's code into executable instructions for a target platform.
4. Develop a standard library of functions and data structures to provide essential functionality to users.
5. Test and refine the language by writing sample programs and gathering user feedback for further improvements.

**Flowchart:**



## Code

|  |
| --- |
| import re  def calculate\_string\_operation(input\_string):  # Define a regular expression pattern to match the input format.  pattern = r'(\d+)\s\*(plus|minus|multiply|divide|mod)\s\*(\d+)'   # Use regex to extract the numbers and operation from the input.  match = re.match(pattern, input\_string)   if match:  # Extract the numbers and operation.  num1 = int(match.group(1))  operation = match.group(2)  num2 = int(match.group(3))   # Perform the specified operation and return the result.  if operation == "plus":  result = num1 + num2  elif operation == "minus":  result = num1 - num2  elif operation == "multiply":  result = num1 \* num2  elif operation == "divide":  if num2 == 0:  return "Division by zero is not allowed."  result = num1 / num2  elif operation == "mod":  if num2 == 0:  return "Modulo by zero is not allowed."  result = num1 % num2  else:  return "Invalid operation."   return result  else:  return "Invalid input format. Please use 'number operation number' format." # Performing Addition input\_string = "6 plus 4" result = calculate\_string\_operation(input\_string) print("6 plus 4 =", result) # Performing Subtraction input\_string = "6 minus 4" result = calculate\_string\_operation(input\_string) print("6 minus 4 =", result)  # Performing Multiplication input\_string = "6 multiply 4" result = calculate\_string\_operation(input\_string) print("6 multiply 4 =",result)  # Performing Division input\_string = "6 divide 4" result = calculate\_string\_operation(input\_string) print("6 divide 4 =", result)  # Performing Mod Operation input\_string = "6 mod 4" result = calculate\_string\_operation(input\_string) print("6 mod 4 =", result)  # Performing Undefinded Operation input\_string = "6 add 4" result = calculate\_string\_operation(input\_string) print("6 add 4 =", result) |

## Output

|  |
| --- |
| 6 plus 4 = 10 6 minus 4 = 2 6 multiply 4 = 24 6 divide 4 = 1.5 6 add 4 = Invalid input format. Please use 'number operation number' format. |

# Experiment 3

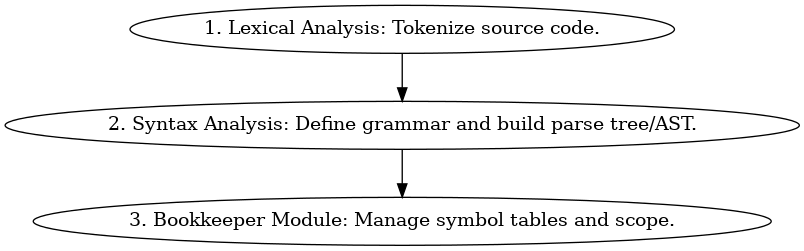
## Aim

Develop a lexical analyzer and a syntax analyzer for the same using the LEX and YACC tools. Also implement the bookkeeper module.

**Steps:**

1. Develop a lexical analyzer using Lex (or similar tool) to tokenize the input source code and generate tokens.
2. Create a syntax analyzer using Yacc (Bison) to define the grammar and build a parse tree or AST for the language.
3. Implement a bookkeeper module to manage symbol tables and support variable declaration, scope management, and symbol resolution.

**Flowchart:**



## Code

### lexer.l

|  |
| --- |
| %{ #include "y.tab.h" #include "bookkeeper.h" %}  %%  [ \t] ; // Ignore whitespace [0-9]+ { yylval.num = atoi(yytext); return NUM; } [a-zA-Z][a-zA-Z0-9]\* { yylval.id = strdup(yytext); return ID; } "=" { return ASSIGN; } "+" { return PLUS; } "-" { return MINUS; } "\*" { return MULTIPLY; } "/" { return DIVIDE; } \n { return EOL; } . { yyerror("Invalid character"); }  %%  int yywrap() {  return 1; } |

### parser.y

|  |
| --- |
| %{ #include <stdio.h> #include <stdlib.h> #include "bookkeeper.h" %}  %token NUM ID ASSIGN PLUS MINUS MULTIPLY DIVIDE EOL  %%  stmt: ID ASSIGN expr EOL { assign\_variable($1, $3); }  | expr EOL { printf("Result: %d\n", $1); }  ;  expr: ID { $$ = get\_variable\_value($1); }  | NUM { $$ = $1; }  | expr PLUS expr { $$ = $1 + $3; }  | expr MINUS expr { $$ = $1 - $3; }  | expr MULTIPLY expr { $$ = $1 \* $3; }  | expr DIVIDE expr { $$ = $1 / $3; }  ;  %%  int main() {  yyparse();  return 0; }  void yyerror(const char\* msg) {  fprintf(stderr, "Error: %s\n", msg);  exit(1); } |

### bookkeeper.c

|  |
| --- |
| #include <stdio.h> #include <stdlib.h> #include <string.h>  typedef struct {  char\* name;  int value; } Variable;  Variable variables[100];  void **assign\_variable**(char\* name, int value) {  for (int i = 0; i < 100; ++i) {  if (variables[i].name == NULL) {  // Found an empty slot, store the variable  variables[i].name = strdup(name);  variables[i].value = value;  printf("Variable %s assigned the value: %d\n", name, value);  return;  } else if (strcmp(variables[i].name, name) == 0) {  // Variable already exists, update its value  variables[i].value = value;  printf("Variable %s updated with the value: %d\n", name, value);  return;  }  }  // No empty slot found, and maximum variables limit reached  printf("Error: Maximum variables limit reached\n");  exit(1); }   int **get\_variable\_value**(char\* name) {  for (int i = 0; i < 100; ++i) {  if (variables[i].name != NULL && strcmp(variables[i].name, name) == 0) {  // Variable found, return its value  return variables[i].value;  }  }  // Variable not found  printf("Error: Variable %s not found\n", name);  exit(1); } |

### bookkeeper.h

|  |
| --- |
| #ifndef BOOKKEEPER\_H #define BOOKKEEPER\_H  void **assign\_variable**(char\* name, int value); int **get\_variable\_value**(char\* name);  #endif |

## 

## Output



# 

# Experiment 4 x

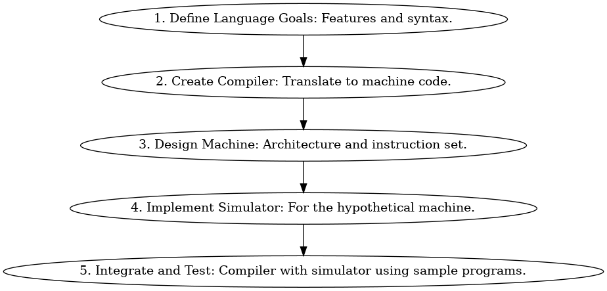
## Aim

﻿﻿﻿Design a small high-level language and implement a compiler for the same. If the target machine of the compiler is a hypothetical machine, then implement a simulator for it.

**Steps:**

1. Define the high-level language's goals, features, and syntax.
2. Create a compiler that translates the high-level language into machine code for a hypothetical target machine.
3. Design the architecture and instruction set for the hypothetical machine.
4. Implement a simulator for the hypothetical machine.

**Flowchart:**

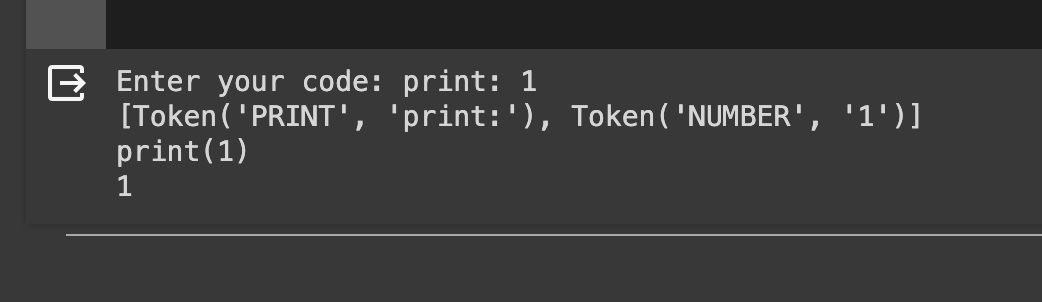


## Code

### main.py

|  |
| --- |
| # Commented out IPython magic to ensure Python compatibility. # %pip install ply rply  from rply import LexerGenerator  lg = LexerGenerator()  # Define token rules lg.add("PRINT", r"print:") lg.add("NUMBER", r"\d+") lg.add("PLUS", r"\+") lg.add("MINUS", r"-") lg.add("MULTIPLY", r"\\*") lg.add("DIVIDE", r"/") lg.add("LPAREN", r"\(") lg.add("RPAREN", r"\)")  # Ignore whitespace lg.ignore(r"\s+")  lexer = lg.build()  # Tokenize a string def **tokenize**(text):  return (lexer.lex(text))  def **get\_tokens**(text):  return list(lexer.lex(text))  class **Node**:  def **accept**(self, visitor):  return visitor.visit(self)  class **Number**(Node):  def **\_\_init\_\_**(self, value):  self.value = value  class **BinaryOp**(Node):  def **\_\_init\_\_**(self, children):  self.left = children[0]  self.right = children[1]  class **Add**(BinaryOp):  def **gettokentype**(self):  return "ADD"  class **Subtract**(BinaryOp):  def **gettokentype**(self):  return "MINUS"  class **Multiply**(BinaryOp):  def **gettokentype**(self):  return "MULTIPLY"  class **Divide**(BinaryOp):  def **gettokentype**(self):  return "DIVIDE"  class **Print**(Node):  def **\_\_init\_\_**(self, children):  self.expression = children[0]  from rply import ParserGenerator pg = ParserGenerator(  ["PRINT", "NUMBER", "PLUS", "MINUS", "MULTIPLY", "DIVIDE", "LPAREN", "RPAREN"] )   @pg.production("program : statement") @pg.production("program : program statement") def **program**(p):  if len(p) == 1:  return p[0]  else:  return Print([p[0], p[1]])  @pg.production("statement : PRINT expression") def **statement\_print**(p):  return Print([p[1]])  @pg.production("statement : expression") def **statement\_expression**(p):  return p[0]  @pg.production("expression : NUMBER") def **expression\_number**(p):  return Number(int(p[0].getstr()))  @pg.production("expression : expression PLUS expression") @pg.production("expression : expression MINUS expression") @pg.production("expression : expression MULTIPLY expression") @pg.production("expression : expression DIVIDE expression") @pg.production("expression : LPAREN expression RPAREN") def **expression\_binop**(p):  left = p[0]   operator = p[1].gettokentype()  right = p[2]  if operator == "PLUS":  return Add([left, right])  elif operator == "MINUS":  return Subtract([left, right])  elif operator == "MULTIPLY":  return Multiply([left, right])  elif operator == "DIVIDE":  return Divide([left, right])  parser = pg.build()  class **CodeGenerator**:  def **visit**(self, node):  method\_name = f"visit\_{type(node).\_\_name\_\_}"  visitor = getattr(self, method\_name, self.generic\_visit)  return visitor(node)   def **generic\_visit**(self, node):  raise Exception(f"No visit\_{type(node).\_\_name\_\_} method")   def **visit\_Number**(self, node):  return str(node.value)   def **visit\_Add**(self, node):  left = self.visit(node.left)  right = self.visit(node.right)  return f"({left} + {right})"   def **visit\_Subtract**(self, node):  left = self.visit(node.left)  right = self.visit(node.right)  return f"({left} - {right})"   def **visit\_Multiply**(self, node):  left = self.visit(node.left)  right = self.visit(node.right)  return f"({left} \* {right})"   def **visit\_Divide**(self, node):  left = self.visit(node.left)  right = self.visit(node.right)  return f"({left} / {right})"   def **visit\_Print**(self, node):  expression = self.visit(node.expression)  return f"print({expression})"  def **main**():  source\_code = input("Enter your code: ")  tokens = tokenize(source\_code)  print(get\_tokens(source\_code))  ast = parser.parse(tokens)  if ast:  code\_generator = CodeGenerator()  python\_code = code\_generator.visit(ast)  print(python\_code)   # Execute the generated Python code  try:  exec(python\_code)  except Exception as e:  print("Error:", e)  else:  print("Parsing failed.")  if \_\_name\_\_ == "\_\_main\_\_":  main() |

## Output



# Experiment 6

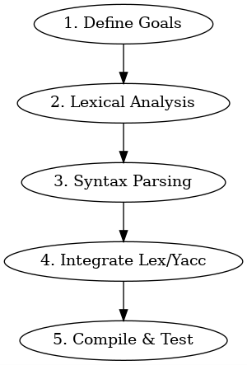
## Aim

Develop a simple calculator using LEX and YACC tools.

**Steps:**

1. Define the calculator's goals and features, specifying arithmetic operations it should support.
2. Create a Lexical Analyzer (Lex) to tokenize input expressions.
3. Develop a Syntax Analyzer (Yacc/Bison) to parse and evaluate arithmetic expressions.
4. Integrate Lex and Yacc to create a complete calculator parser.
5. Compile the parser and test it with sample arithmetic expressions to ensure correct evaluation.

**Flowchart:**



## Code

### LEX PART:

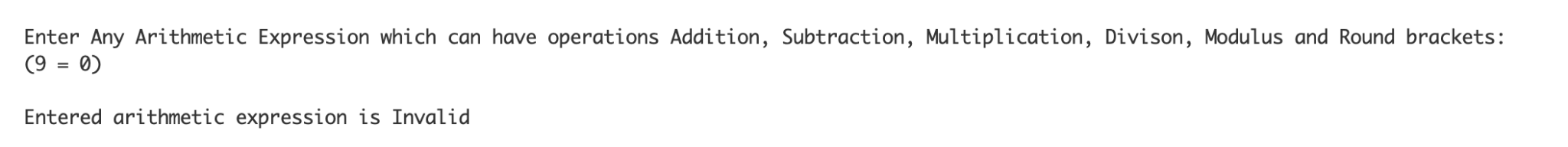
|  |
| --- |
| %{ #include<stdio.h> #include "y.tab.h" extern int yylval; %}    %% [0-9]+ {  yylval=atoi(yytext);  return NUMBER; }  [\t] ;  [\n] return 0;  . return yytext[0]; %%  int **yywrap**() { return 1; } |

### YACC

|  |
| --- |
| %{  #include<stdio.h>  int flag=0;  int yylex(void);  void yyerror(); %}  %token NUMBER %left '+' '-' %left '\*' '/' '%' %left '(' ')' %%  ArithmeticExpression: E{  printf("\nResult=%d\n",$$);  return 0; };  E:E'+'E {$$=$1+$3;}  |E'-'E {$$=$1-$3;}  |E'\*'E {$$=$1\*$3;}  |E'/'E {$$=$1/$3;}  |E'%'E {$$=$1%$3;}  |'('E')' {$$=$2;}  | NUMBER {$$=$1;} ; %%  void **main**() {  printf("\nEnter Any Arithmetic Expression which can have operations Addition, Subtraction, Multiplication, Divison, Modulus and Round brackets:\n");  yyparse();   if(flag==0)  printf("\nEntered arithmetic expression is Valid\n\n"); }  void **yyerror**() {  printf("\nEntered arithmetic expression is Invalid\n\n");  flag=1; } |

## 

## Output



# 

# Experiment 7

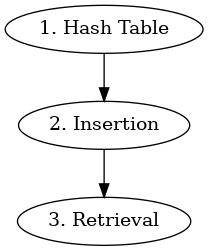
## Aim

Implement a program for symbol table using hashing

**Steps:**

1. Create a hash table structure with a hash function.
2. Implement an insertion function to add identifiers and attributes to the hash table.
3. Develop a retrieval function to look up identifiers and return their attributes.

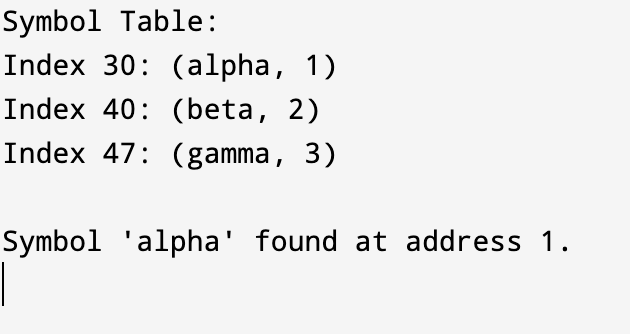
**Flowchart:**



## Code

|  |
| --- |
| #include <stdio.h> #include <stdlib.h> #include <string.h>  #define TABLE\_SIZE 100  // Node structure for linked list typedef struct **Node** {  char\* symbol;  int address;  struct **Node**\* **next**; } Node;  // Symbol table as an array of linked lists (chains) Node\* symbolTable[TABLE\_SIZE];  // Hash function unsigned int **hash**(char\* symbol) {  unsigned int value = 0;  for (; \*symbol; ++symbol) {  value = value \* 33 + \*symbol; // A simple hash function  }  return value % TABLE\_SIZE; }  // Function to insert a symbol void **insert**(char\* symbol, int address) {  unsigned int index = hash(symbol);  Node\* newNode = (Node\*)malloc(sizeof(Node));  newNode->symbol = strdup(symbol); // Duplicate the symbol for storage  newNode->address = address;  newNode->next = symbolTable[index];  symbolTable[index] = newNode; }  // Function to find a symbol Node\* **find**(char\* symbol) {  unsigned int index = hash(symbol);  Node\* currentNode = symbolTable[index];  while (currentNode) {  if (strcmp(currentNode->symbol, symbol) == 0) {  return currentNode;  }  currentNode = currentNode->next;  }  return NULL; // Not found }  // Function to display the symbol table void **display**() {  printf("Symbol Table:\n");  for (int i = 0; i < TABLE\_SIZE; ++i) {  Node\* currentNode = symbolTable[i];  if (currentNode) {  printf("Index %d: ", i);  while (currentNode) {  printf("(%s, %d) ", currentNode->symbol, currentNode->address);  currentNode = currentNode->next;  }  printf("\n");  }  } }  // Main function int **main**() {  // Initialize the symbol table  memset(symbolTable, 0, sizeof(symbolTable));   // Insert some symbols  insert("alpha", 1);  insert("beta", 2);  insert("gamma", 3);   // Display the symbol table  display();   // Find a symbol  char\* searchSymbol = "alpha";  Node\* foundNode = find(searchSymbol);  if (foundNode) {  printf("\nSymbol '%s' found at address %d.\n", foundNode->symbol, foundNode->address);  } else {  printf("\nSymbol '%s' not found.\n", searchSymbol);  }   // Cleanup memory (not shown here)   return 0; } |

## Output



# 

# 

# 

# 

# Experiment 7

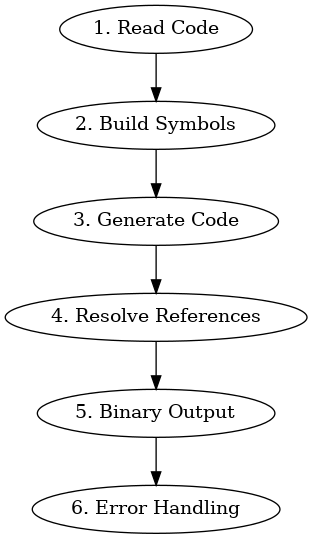
## Aim

Two pass assembler

**Steps:**

1. In the first pass, read the source code to build a symbol table, assigning addresses to labels and recording variable declarations.
2. In the second pass, generate machine code instructions while resolving symbolic references using the symbol table.
3. Create a binary machine code output file with assembled instructions.
4. Optionally, handle error checking and reporting during both passes to ensure valid assembly code.

**Flowchart:**



## Code

|  |
| --- |
| #include <stdio.h> #include <string.h> #include <stdlib.h>  void **chk\_label**(); void **chk\_opcode**(); void **READ\_LINE**(); struct **optab** {  char code[10], objcode[10]; } myoptab[3] = {  {"LDA", "00"},  {"JMP", "01"},  {"STA", "02"}};  struct **symtab** {  char symbol[10];  int addr; } mysymtab[10];  int startaddr, locctr, symcount = 0, length; char line[20], label[8], opcode[8], operand[8], programname[10];  void **create\_input\_file**() {  FILE \*input;  input = fopen("input.txt", "w");  if (input == NULL)  {  printf("Error creating input.txt\n");  exit(1);  }   fprintf(input, "MYPGM START 1000\n");  fprintf(input, "STA\n");  fprintf(input, "LOOP1 JMP LOOP2\n");  fprintf(input, "LDA\n");  fprintf(input, "LOOP2 JMP LOOP1\n");  fprintf(input, "RESB 04\n");  fprintf(input, "LDA\n");  fprintf(input, "STA\n");  fprintf(input, "JMP LOOP1\n");  fprintf(input, "END");   fclose(input); }  void **PASS1**() {  FILE \*input, \*inter;  input = fopen("input.txt", "r");  inter = fopen("inter.txt", "w");  printf("LOCATION LABEL\tOPERAND\tOPCODE\n");  printf("\_\_\_\_\_\_\_\_\_\_\_\_\_\n");  fgets(line, 20, input);  READ\_LINE();  if (!strcmp(opcode, "START"))  {  startaddr = atoi(operand);  locctr = startaddr;  strcpy(programname, label);  fprintf(inter, "%s", line);  fgets(line, 20, input);  }  else  {  programname[0] = '\0';  startaddr = 0;  locctr = 0;  }  printf("\n %04X\t %s\t%s\t %s", locctr, label, opcode, operand);  while (strcmp(line, "END") != 0)  {  READ\_LINE();  printf("\n %04X\t %s\t%s\t %s", locctr, label, opcode, operand);  if (label[0] != '\0')  chk\_label();  chk\_opcode();  fprintf(inter, "%s %s %s\n", label, opcode, operand);  fgets(line, 20, input);  }  printf("\n %04X\t\t%s", locctr, line);  fprintf(inter, "%s", line);  fclose(inter);  fclose(input); }  void **PASS2**() {  FILE \*inter, \*output;  char record[70], part[10], value[10];  int currtxtlen = 0, foundopcode, foundoperand, chk, operandaddr, recaddr = 0;  inter = fopen("inter.txt", "r");  output = fopen("output.txt", "w");  fgets(line, 20, inter);  READ\_LINE();  if (!strcmp(opcode, "START"))  fgets(line, 20, inter);  printf("\n\nCorresponding Object code is..\n");  printf("\nH^ %s ^ %04X ^ %04X ", programname, startaddr, length);  fprintf(output, "\nH^ %s ^ %04X ^ %04X ", programname, startaddr, length);  recaddr = startaddr;  record[0] = '\0';  while (strcmp(line, "END") != 0)  {  operandaddr = foundoperand = foundopcode = 0;  value[0] = part[0] = '\0';  READ\_LINE();  for (chk = 0; chk < 3; chk++)  {  if (!strcmp(opcode, myoptab[chk].code))  {  foundopcode = 1;  strcpy(part, myoptab[chk].objcode);  if (operand[0] != '\0')  {  for (chk = 0; chk < symcount; chk++)  {  if (!strcmp(mysymtab[chk].symbol, operand))  {  sprintf(value, "%04X", mysymtab[chk].addr);  strcat(part, value);  foundoperand = 1;  }  }  if (!foundoperand)  strcat(part, "err");  }  }  }  if (!foundopcode)  {  if (strcmp(opcode, "BYTE") == 0 || strcmp(opcode, "WORD") == 0 || strcmp(opcode, "RESB") == 0)  {  strcat(part, operand);  }  }  if ((currtxtlen + strlen(part)) <= 8)  {  strcat(record, "^");  strcat(record, part);  currtxtlen += strlen(part);  }  else  {  printf("\nT^ %04X ^%02X %s", recaddr, currtxtlen / 2, record);  fprintf(output, "\nT^ %04X ^%02X %s", recaddr, currtxtlen / 2, record);  recaddr += currtxtlen / 2;  currtxtlen = strlen(part);  strcpy(record, part);  }  fgets(line, 20, inter);  }  printf("\nT^ %04X ^%02X %s", recaddr, currtxtlen / 2, record);  fprintf(output, "\nT^ %04X ^%02X %s", recaddr, currtxtlen / 2, record);  printf("\nE^ %04X\n", startaddr);  fprintf(output, "\nE^ %04X\n", startaddr);  fclose(inter);  fclose(output); }  void **READ\_LINE**() {  char buff[8], word1[8], word2[8], word3[8];  int i, j = 0, count = 0;  label[0] = opcode[0] = operand[0] = word1[0] = word2[0] = word3[0] = '\0';  for (i = 0; line[i] != '\0'; i++)  {  if (line[i] != ' ')  buff[j++] = line[i];  else  {  buff[j] = '\0';  strcpy(word3, word2);  strcpy(word2, word1);  strcpy(word1, buff);  j = 0;  count++;  }  }  buff[j - 1] = '\0';  strcpy(word3, word2);  strcpy(word2, word1);  strcpy(word1, buff);  switch (count)  {  case 0:  strcpy(opcode, word1);  break;  case 1:  strcpy(opcode, word2);  strcpy(operand, word1);  break;  case 2:  strcpy(label, word3);  strcpy(opcode, word2);  strcpy(operand, word1);  break;  } }  void **chk\_label**() {  int k, dupsym = 0;  for (k = 0; k < symcount; k++)  {  if (!strcmp(label, mysymtab[k].symbol))  {  mysymtab[k].addr = -1;  dupsym = 1;  break;  }  }  if (!dupsym)  {  strcpy(mysymtab[symcount].symbol, label);  mysymtab[symcount++].addr = locctr;  } }  void **chk\_opcode**() {  int k = 0, found = 0;  for (k = 0; k < 3; k++)  {  if (!strcmp(opcode, myoptab[k].code))  {  locctr += 3;  found = 1;  break;  }  }  if (!found)  {  if (!strcmp(opcode, "WORD"))  locctr += 3;  else if (!strcmp(opcode, "RESW"))  locctr += (3 \* atoi(operand));  else if (!strcmp(opcode, "RESB"))  locctr += atoi(operand);  } }  int **main**() {  create\_input\_file();  PASS1();  length = locctr - startaddr;  PASS2();  return 0; } Output |

# Experiment 8

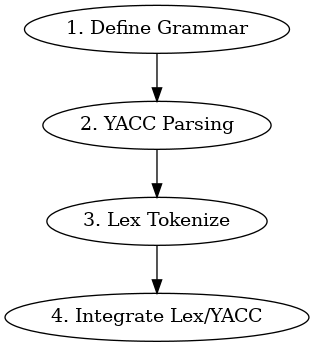
## Aim

Implement a bottom-up parser using the YACC tool.

**Steps:**

1. Define the grammar for the language, specifying the rules and productions.
2. Use YACC (Bison) to create parser rules and actions for each grammar rule, implementing a bottom-up parsing strategy.
3. Write a Lexical Analyzer (Lex) to tokenize the input code.
4. Integrate the Lexical Analyzer with the YACC-generated parser to process and parse the input code according to the defined grammar.

**Flowchart:**



## Procedure

### Step 1: Write the Lexical Analyzer (lexer.l) using Flex:

1. Create a new file called lexer.l.
2. Define regular expressions to match tokens (e.g., numbers, operators) and provide corresponding actions.
3. Include the necessary header files and define the main function.
4. Compile the lexer using lex:

|  |
| --- |
| lex lexer.l |

### Step 2: Write the Grammar (parser.y) using YACC:

1. Create a new file called parser.y.
2. Define the grammar rules and associated actions using YACC syntax.
3. Include necessary header files and define the main function.
4. Generate parser files (y.tab.c and y.tab.h) using yacc:

|  |
| --- |
| yacc -d parser.y |

### Step 3: Compile and Link the Code:

|  |
| --- |
| $ lex lexer.l # Generates lex.yy.c $ yacc -d parser.y # Generates y.tab.c and y.tab.h $ gcc lex.yy.c y.tab.c -o parser -lm |

## Code

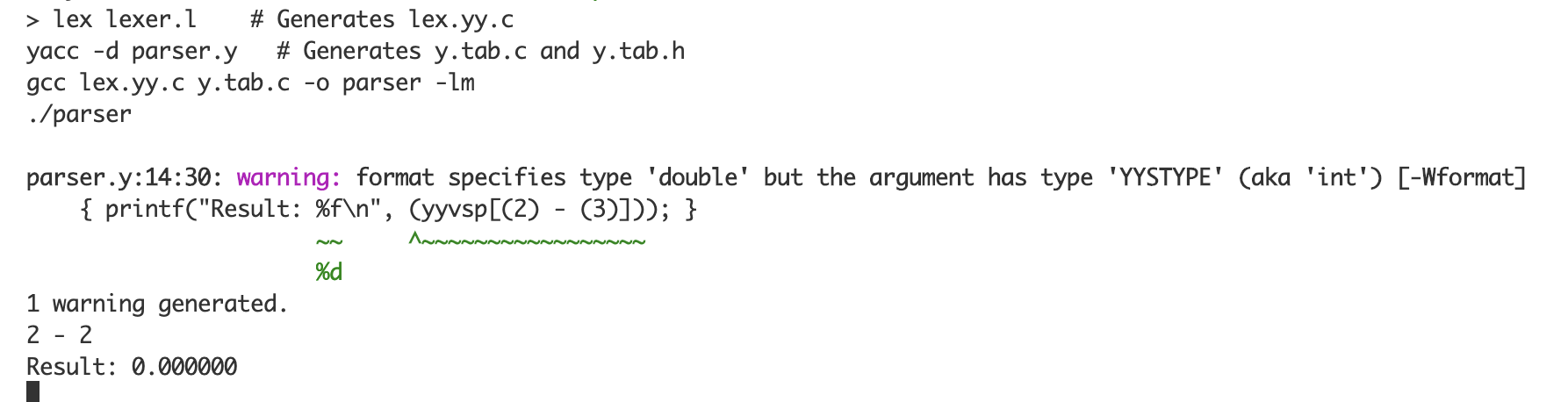
### parser.y

|  |
| --- |
| %{ #include <stdio.h>  int yylex(void);  int yyerror(const char\* s); %}  %token NUMBER %left '+' '-' %left '\*' '/'  %%  input: /\* empty \*/  | input expr '\n' { printf("Result: %f\n", $2); }  ;  expr: NUMBER { $$ = $1; }  | expr '+' expr { $$ = $1 + $3; }  | expr '-' expr { $$ = $1 - $3; }  | expr '\*' expr { $$ = $1 \* $3; }  | expr '/' expr { $$ = $1 / $3; }  | '(' expr ')' { $$ = $2; }  ;  %%  int **main**() {  yyparse();  return 0; }  int yyerror(const char \*s) {  fprintf(stderr, "Error: %s\n", s);  return 0; } |

### lexer.l

|  |
| --- |
| %{ #include "y.tab.h" %}  %% [0-9]+(\.[0-9]+)? {  yylval = atof(yytext);  return NUMBER; } [-+\*/()\n] {  return yytext[0]; } [ \t]+ {  // Ignore spaces and tabs } . {  fprintf(stderr, "Error: Invalid character '%c'\n", yytext[0]); } %%  int **yywrap**() {  return 1; } |

## Output



# Experiment 9

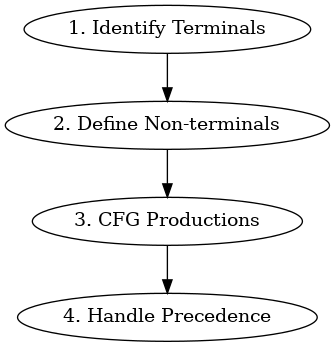
## Aim

Represent 'C' language using Context Free Grammar

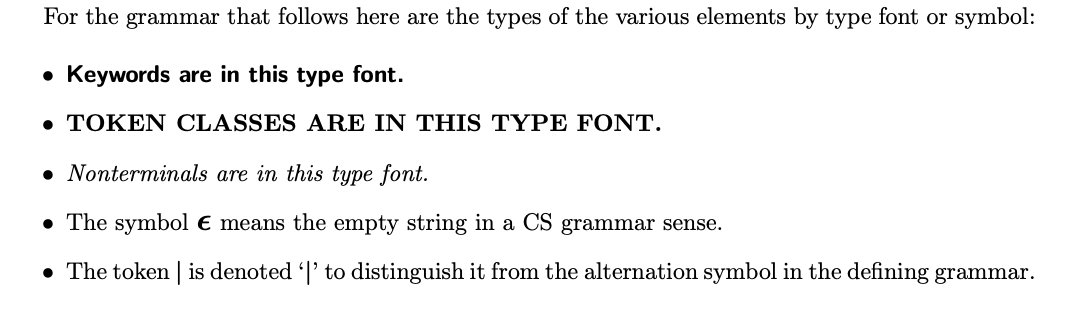
**Steps:**

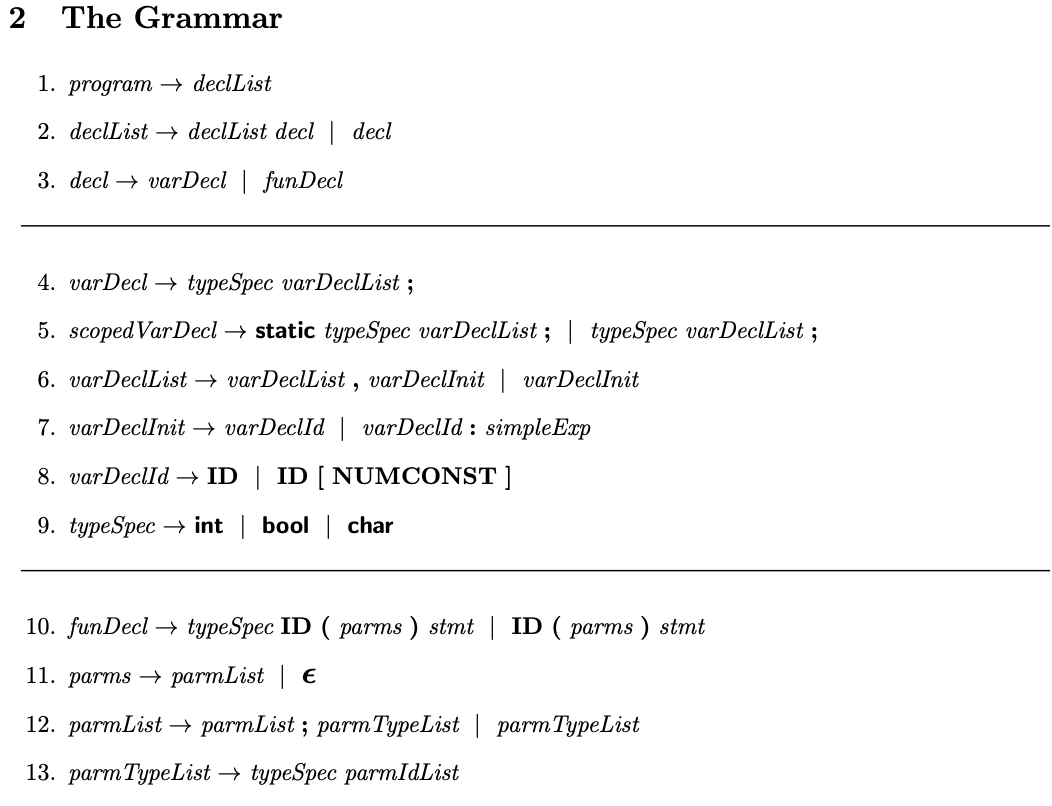
1. Identify terminals (e.g., keywords, symbols) and non-terminals (e.g., expressions, statements) in the C language.
2. Define CFG productions that describe how language constructs are formed, starting with fundamental elements and building complexity.
3. Handle operator precedence and associativity by incorporating these rules into the grammar.

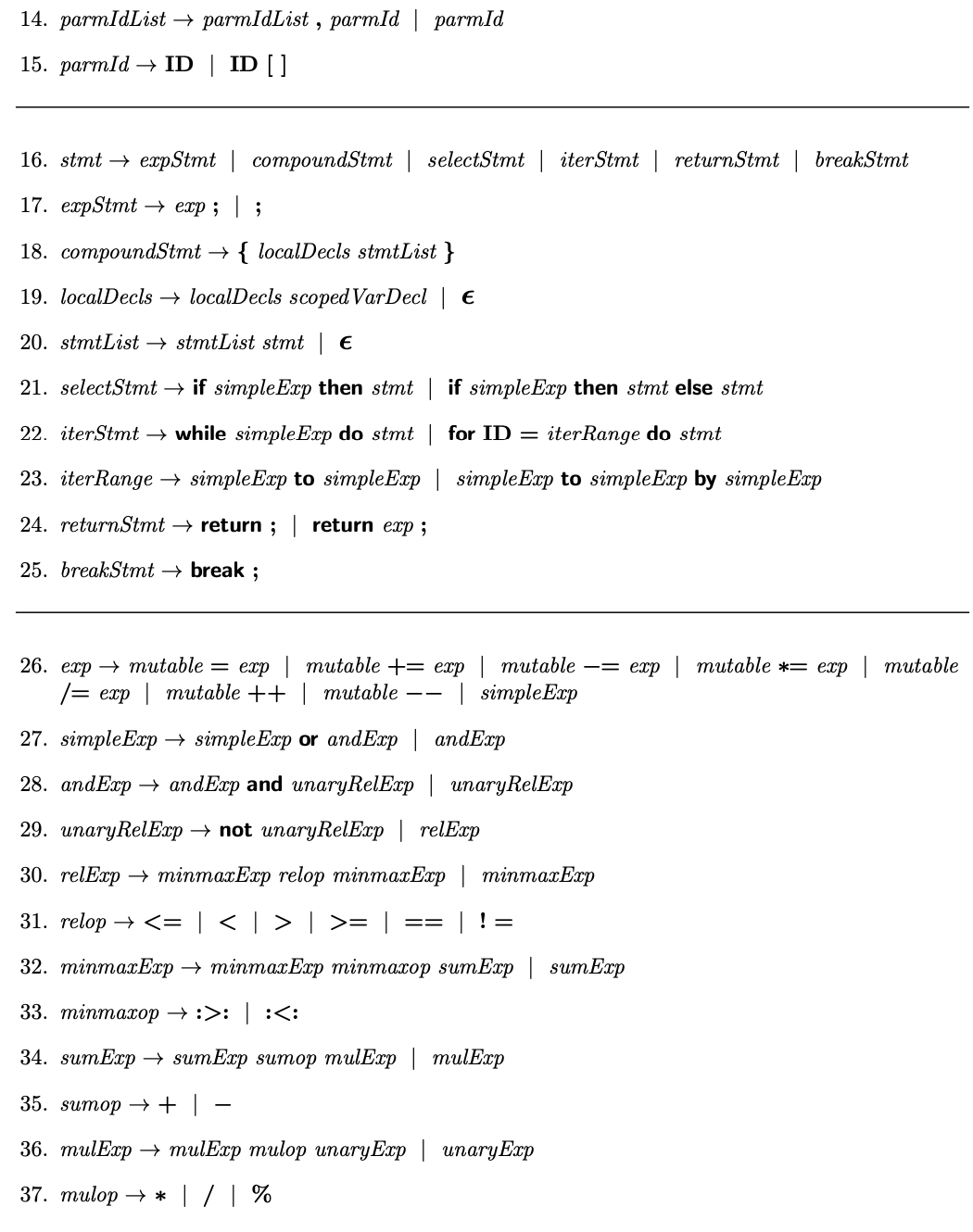
**Flowchart**

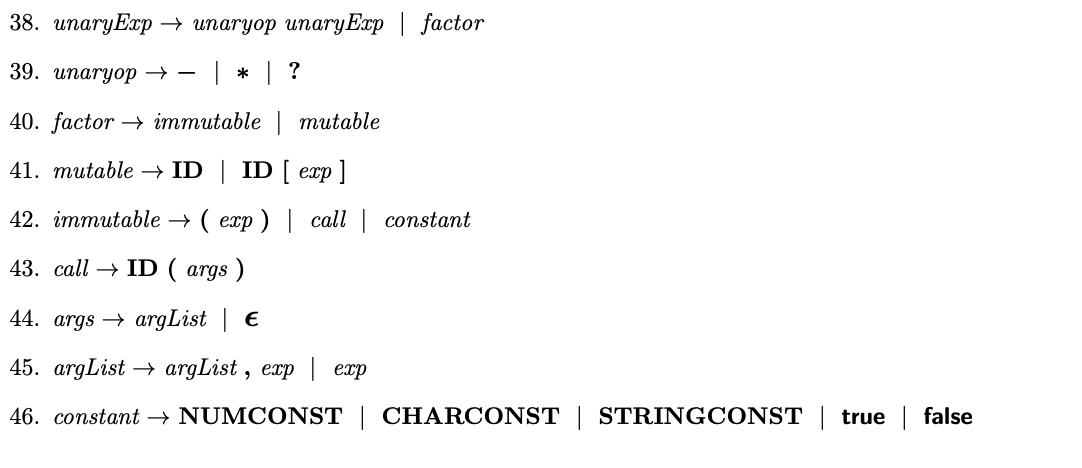


## Code









# Experiment 10

## Aim

Add assignment statement, If then else statement and while loop to the calculator and generate the three address code for the same

**Steps:**

1. Extend the calculator's grammar to include assignment statements, "if-then-else" statements, and while loops.
2. Modify the parser actions to generate three-address code for these new language constructs.
3. Implement a data structure to store and manage the three-address code instructions.
4. Parse and evaluate a sample program that includes these new constructs, producing the corresponding three-address code.
5. Display the generated three-address code as the output.

**Flowchart:**



## Code

|  |
| --- |
| #include <stdio.h> #include <math.h> #include <stdbool.h>  double variables[26]; // Store variables a to z int temp\_var\_count = 0; // Counter for generating temporary variable names  void evaluate\_expression(char \*expression) {  double result;  if (sscanf(expression, "%lf", &result) == 1) {  printf("Result: %lf\n", result);  } else {  printf("Error: Invalid expression\n");  } }  char\* **generate\_temp\_var**() {  static char temp\_var\_name[3]; // Assuming a maximum of 100 temporary variables  snprintf(temp\_var\_name, sizeof(temp\_var\_name), "t%d", temp\_var\_count);  temp\_var\_count++;  return temp\_var\_name; }  void assign\_variable(char variable, double value) {  variables[variable - 'a'] = value; }  void generate\_assignment\_code(char \*assignment) {  char variable;  double value;  if (sscanf(assignment, " %c = %lf", &variable, &value) == 2) {  assign\_variable(variable, value);  printf("%c = %lf\n", variable, value);  } else {  printf("Error: Invalid assignment statement\n");  } }  void generate\_if\_else\_code(char \*if\_condition, char \*if\_body, char \*else\_body) {  double condition\_value;  if (sscanf(if\_condition, " %lf", &condition\_value) == 1) {  if (condition\_value != 0.0) {  printf("Executing if body: %s\n", if\_body);  } else {  printf("Executing else body: %s\n", else\_body);  }  } else {  printf("Error: Invalid if condition\n");  } }  void generate\_while\_loop\_code(char \*while\_condition, char \*loop\_body) {  double condition\_value;  if (sscanf(while\_condition, " %lf", &condition\_value) == 1) {  while (condition\_value != 0.0) {  printf("Executing loop body: %s\n", loop\_body);  if (sscanf(while\_condition, " %lf", &condition\_value) != 1) {  printf("Error: Invalid while condition\n");  break;  }  }  } else {  printf("Error: Invalid while condition\n");  } }  int **main**() {  char input[100];  while (true) {  printf("Select operation (calculator, assignment, ifelse, whileloop, or exit): ");  fgets(input, sizeof(input), stdin);  if (input[0] == 'e' && input[1] == 'x' && input[2] == 'i' && input[3] == 't') {  break;  } else if (input[0] == 'c' && input[1] == 'a' && input[2] == 'l' && input[3] == 'c') {  printf("Enter an expression: ");  fgets(input, sizeof(input), stdin);  evaluate\_expression(input);  } else if (input[0] == 'a' && input[1] == 's' && input[2] == 's') {  printf("Enter an assignment statement (e.g., x = 5): ");  fgets(input, sizeof(input), stdin);  generate\_assignment\_code(input);  } else if (input[0] == 'i' && input[1] == 'f') {  printf("Enter if condition (e.g., 2): ");  fgets(input, sizeof(input), stdin);  char if\_condition[100];  strcpy(if\_condition, input);   printf("Enter if body: ");  fgets(input, sizeof(input), stdin);  char if\_body[100];  strcpy(if\_body, input);   printf("Enter else body: ");  fgets(input, sizeof(input), stdin);  char else\_body[100];  strcpy(else\_body, input);   generate\_if\_else\_code(if\_condition, if\_body, else\_body);  } else if (input[0] == 'w' && input[1] == 'h' && input[2] == 'i') {  printf("Enter while condition (e.g., i): ");  fgets(input, sizeof(input), stdin);  char while\_condition[100];  strcpy(while\_condition, input);   printf("Enter loop body: ");  fgets(input, sizeof(input), stdin);  char loop\_body[100];  strcpy(loop\_body, input);   generate\_while\_loop\_code(while\_condition, loop\_body);  } else {  printf("Invalid choice. Please select calculator, assignment, ifelse, whileloop, or exit.\n");  }  }  return 0; } |