Expression Evaluator:

Data Structures: *Stack (linked list) for operands, linked list (or tree) for the expression tree (optional).*

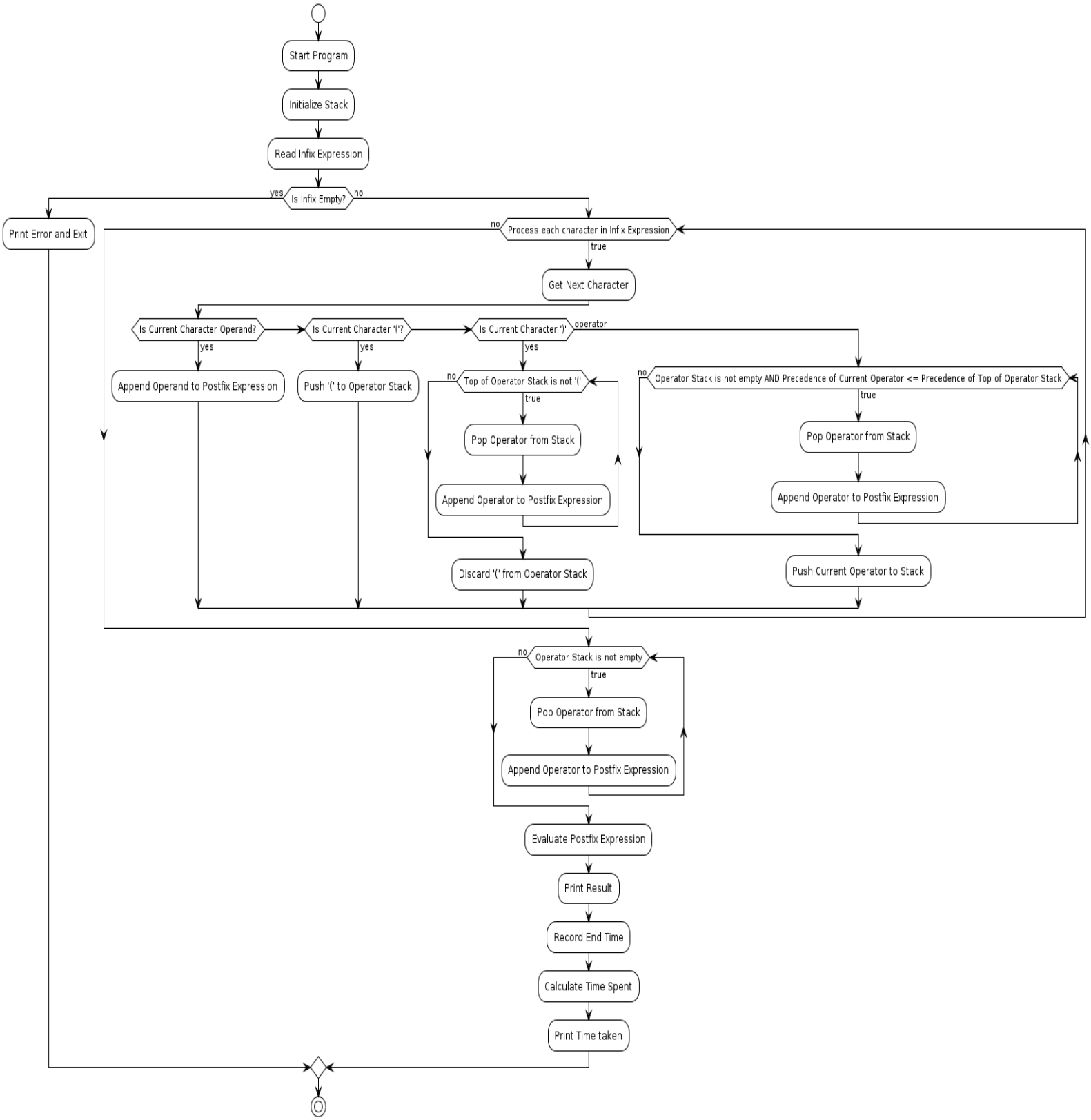
Functionality:

*Allow users to enter infix expressions (e.g., (a + b) * c).*

Convert the expression to postfix (Polish notation) using a shunting-yard algorithm.

*Evaluate the postfix expression using a stack, supporting basic arithmetic operations (+, -, *, /).*

Consider extending to more complex expressions with parentheses and functions.


Start Program
Initialize Stack
Read Infix Expression
Is Infix Empty?
yes
no
Print Error and Exit
Process each character in Infix Expression
no
true
Get Next Character
Is Current Character Operand?
Is Current Character '('?
Is Current Character ')'
operator
yes
Append Operand to Postfix Expression
Push '(' to Operator Stack
Top of Operator Stack is not '('
Pop Operator from Stack
Append Operator to Postfix Expression
Discard '(' from Operator Stack
Operator Stack is not empty AND Precedence of Current Operator <= Precedence of Top of Operator Stack
Push Current Operator to Stack
Operator Stack is not empty
Evaluate Postfix Expression
Print Result
Record End Time
Calculate Time Spent
Print Time taken

evalapi.c

```
#include "func.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

StackNode* createStackNode(int data) {
    StackNode* newNode = (StackNode*)malloc(sizeof(StackNode));
    if (newNode == NULL) {
        printf("Memory allocation failed.\n");
        exit(EXIT_FAILURE);
    }
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

void push(StackNode** top, int data) {
    StackNode* newNode = createStackNode(data);
    newNode->next = *top;
    *top = newNode;
}

int pop(StackNode** top) {
    if (isEmpty(*top)) {
        printf("Stack underflow.\n");
        exit(EXIT_FAILURE);
    }
```

```
    StackNode* temp = *top;
    int data = temp->data;
    *top = (*top)->next;
    free(temp);
    return data;
}

int isEmpty(StackNode* top) {
    return top == NULL;
}

int peek(StackNode* top) {
    if (isEmpty(top)) {
        printf("Stack is empty.\n");
        exit(EXIT_FAILURE);
    }
    return top->data;
}

int isOperator(char ch) {
    return (ch == '+' || ch == '-' || ch == '*' || ch == '/');
}

int precedence(char op) {
    if (op == '+' || op == '-')
        return 1;
    if (op == '*' || op == '/')
        return 2;
    return 0;
}
```

```
void infixToPostfix(char* infix, char* postfix) {
    StackNode* stack = NULL;
    int i = 0, j = 0;
    while (infix[i]) {
        if (infix[i] == ' ' || infix[i] == '\t') {
            i++;
            continue;
        }
        if (isdigit(infix[i])) {
            postfix[j++] = infix[i++];
            while (isdigit(infix[i]))
                postfix[j++] = infix[i++];
            postfix[j++] = ' ';
        } else if (infix[i] == '(') {
            push(&stack, infix[i++]);
        } else if (infix[i] == ')') {
            while (!isEmpty(stack) && peek(stack) != '(')
                postfix[j++] = pop(&stack);
            if (!isEmpty(stack) && peek(stack) != '(') {
                printf("Invalid expression.\n");
                exit(EXIT_FAILURE);
            } else {
                pop(&stack);
            }
            i++;
        } else if (isOperator(infix[i])) {
            while (!isEmpty(stack) && precedence(infix[i]) <= precedence(peek(stack)))
                postfix[j++] = pop(&stack);
            push(&stack, infix[i++]);
        } else {
            printf("Invalid character in expression.\n");
```

```
            exit(EXIT_FAILURE);
        }
    }
    while (!isEmpty(stack))
        postfix[j++] = pop(&stack);
    postfix[j] = '\0';
}

int evaluatePostfix(char* postfix) {
    StackNode* stack = NULL;
    int i = 0;
    while (postfix[i]) {
        if (postfix[i] == ' ' || postfix[i] == '\t') {
            i++;
            continue;
        }
        if (isdigit(postfix[i])) {
            int operand = 0;
            while (isdigit(postfix[i])) {
                operand = operand * 10 + (postfix[i] - '0');
                i++;
            }
            push(&stack, operand);
        } else if (isOperator(postfix[i])) {
            int operand2 = pop(&stack);
            int operand1 = pop(&stack);
            switch (postfix[i]) {
                case '+': push(&stack, operand1 + operand2); break;
                case '-': push(&stack, operand1 - operand2); break;
                case '*': push(&stack, operand1 * operand2); break;
                case '/': push(&stack, operand1 / operand2); break;
```

```
            }
            i++;
        }
    }
    return pop(&stack);
}
```

eval.h

```
#ifndef FUNC_H
#define FUNC_H

// Structure for a node in the stack
struct StackNode {
    int data;
    struct StackNode* next;
};

typedef struct StackNode StackNode;

// Function declarations
void push(StackNode** top, int data);
int pop(StackNode** top);
int isEmpty(StackNode* top);
int peek(StackNode* top);
int isOperator(char ch);
int precedence(char op);
void infixToPostfix(char* infix, char* postfix);
int evaluatePostfix(char* postfix);

#endif /* FUNC_H */
```

evalmain.c

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "func.h"

int main() {
    clock_t start, end;
    double cpu_time_used;

    char infix[100], postfix[100];

    // Input infix expression
    printf("Enter an infix expression: ");
    fgets(infix, sizeof(infix), stdin);

    // Remove newline character from input
    infix[strcspn(infix, "\n")] = '\0';

    // Convert infix to postfix
    infixToPostfix(infix, postfix);
    printf("Postfix expression: %s\n", postfix);

    // Evaluate postfix expression
    int result = evaluatePostfix(postfix);
    printf("Result: %d\n", result);

    // Measure time taken
    start = clock();
    // Perform operations here...
```

```
    end = clock();
    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
    printf("Time taken: %f seconds\n", cpu_time_used);

    return 0;
}
```

Explanation:

Infix Expressions:

Infix notation is the standard way of writing mathematical expressions where operators are placed between operands. For example, 3 + 4.

*Infix expressions can include parentheses to specify the order of operations, such as (3 + 4) * 5.*

Postfix (Polish Notation):

Postfix notation, also known as Reverse Polish Notation (RPN), places operators after their operands. For example, 3 4 +.

Postfix notation has the advantage of being easily evaluated with a stack-based algorithm, as it doesn't require explicit precedence rules or parentheses.

Shunting-Yard Algorithm:

The Shunting-Yard algorithm converts infix expressions to postfix notation. It was invented by Edsger Dijkstra.

The algorithm uses a stack to keep track of operators and their precedence. It scans the infix expression from left to right, pushing operators onto the stack and outputting operands in postfix notation.

When encountering an operator, the algorithm compares its precedence with the operator at the top of the stack. If the incoming operator has higher precedence, it's pushed onto the stack. If it has lower precedence, operators are popped from the stack and added to the output until the precedence condition is met.

Parentheses are handled specially: when a closing parenthesis is encountered, operators are popped from the stack and added to the output until an opening parenthesis is reached.

Stack-Based Evaluation:

Once the expression is in postfix notation, it can be evaluated using a stack.

Start scanning the postfix expression from left to right.

If an operand is encountered, push it onto the stack.

If an operator is encountered, pop the required number of operands from the stack, perform the operation, and push the result back onto the stack.

After scanning the entire expression, the final result will be on the top of the stack.

Handling More Complex Expressions:

To handle more complex expressions with parentheses and functions, you need to extend the Shunting-Yard algorithm and evaluation process.

Proper handling of parentheses ensures that the order of operations is maintained.

Supporting functions requires parsing and recognizing function calls in the input expression. You would need to parse function names and arguments, evaluate the arguments, and apply the function to the arguments during evaluation.

By combining these elements, you can build a robust expression evaluator that can handle various types of mathematical expressions efficiently.

Output: