N. Sai Kiran
03/05/2024

PROJECT 1:

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

IMPLIMANTATION:

Stack.c

```
#include "stack.h" // Include header file for stack structure definition (assumed)

 #include<stdlib.h> // Include standard library for memory allocation (malloc, free)

#include <stdio.h>// Include standard input/output header (printf)

 // Function to push data onto stack

void push(Stack* stack, char data) {

// Create a new node for the data

 Node* newNode = (Node*) malloc(sizeof(Node));

 if (newNode == NULL) {

 printf("Error: Memory allocation failed\n");
```

```
 exit(EXIT_FAILURE);
}
// Set data and next pointer of the new node
 newNode->data = data;
newNode->next = stack->top;
// Update the top pointer of the stack to point to the new node
 stack->top = newNode;
}
// Function to pop data from stack
 char pop(Stack* stack) {
// Check if stack is empty
if (is_empty(stack)) {
 printf("Error: Stack is empty\n");
exit(EXIT_FAILURE);
}
// Store the top node and its data
Node* temp = stack->top;
 char data = temp->data;
// Update the top pointer to point to the next node
stack->top = temp->next;
// Free the memory used by the popped node
free(temp);
// Return the popped data
return data;
}
// Function to peek at the top of the stack
```

N. Sai Kiran
03/05/2024

```
char peek(Stack* stack) {
 // Check if stack is empty
if (is_empty(stack)) {
 printf("Error: Stack is empty\n");
exit(EXIT_FAILURE);
}
// Return the data of the top node without removing it
return stack->top->data;
}
// Function to check if stack is empty
int is_empty(Stack* stack) {
// Return true if the top pointer is NULL (empty stack)
 return stack->top == NULL;
}
```

EXPLANATION:

This code defines a basic stack data structure and its associated functions in C. Here's a breakdown of each part:

1. **Header Includes**: The code includes necessary header files such as `stack.h` (presumably containing the definition of the stack structure) and standard library headers for memory allocation (`stdlib.h`) and input/output (`stdio.h`).

2. **Function `push`**: This function is used to push data onto the stack. It allocates memory for a new node, assigns the data to it, and updates the top pointer of the stack.

3. **Function `pop`**: This function removes and returns the top element of the stack. It checks if the stack is empty, retrieves the data from the top node, updates the top pointer to point to the next node, frees the memory of the popped node, and returns the data.

N. Sai Kiran
03/05/2024

4. **Function `peek`**: This function returns the value of the top element of the stack without removing it. It checks if the stack is empty and returns the data of the top node.

5. **Function `is_empty`**: This function checks if the stack is empty by verifying if the top pointer is `NULL`. It returns `1` if the stack is empty and `0` otherwise.

Overall, this code provides a basic implementation of a stack data structure in C with common stack operations like push, pop, peek, and checking if the stack is empty.

Stack.h

```
#ifndef STACK_H // Check if STACK_H hasn't been included yet

#define STACK_H // Define header guard to prevent multiple inclusions

// Structure for stack node

typedef struct Node {

char data; // Data element stored in the node (character in this case)

struct Node* next; // Pointer to the next node in the stack

} Node;

 // Structure for stack

typedef struct {

Node* top; // Pointer to the topmost node of the stack

} Stack;

// Function prototypes

void push(Stack* stack, char data);    // Push data onto the stack

char pop(Stack* stack);      // Pop data from the stack

char peek(Stack* stack);    // Peek at the top element without popping

int is_empty(Stack* stack);  // Check if the stack is empty

#endif/* STACK_H */
```

EXPLANATION:

N. Sai Kiran
03/05/2024

This updated version of the code includes a header guard (`#ifndef`, `#define`, and `#endif`) to prevent multiple inclusions of the header file `stack.h`. This ensures that if the header file is included multiple times in the same translation unit, its contents will only be processed once.

The header file also defines the structure for a stack node (`Node`) and a stack (`Stack`). It declares function prototypes for stack operations (`push`, `pop`, `peek`, and `is_empty`), allowing other parts of the program to use these functions without needing to know their implementations.

By encapsulating the stack-related declarations and function prototypes within the header guard, this header file can be safely included in other source files, preventing naming conflicts and ensuring modular code organization.

Expression_tree.c

```
#include "stack.h" // Include header file for stack operations (assumed)

#include <ctype.h> // Include header for character classification functions

#include<stdio.h> // Include standard input/output header

// Function to check if character is an operator

int is_operator(char c) {

// Check if the character is one of the common arithmetic operators

return c == '+' || c == '-' || c == '*' || c == '/';

}

 // Function to get precedence of operator

int precedence(char c) {

// Assign precedence levels based on operator type

if (c == '+' || c == '-') {

return 1;

 } else if (c == '*' || c == '/') {

 return 2;

} else {

return 0; // Parentheses have highest precedence (convention)
```

```
    }
 }
// Function to convert infix expression to postfix
void infix_to_postfix(char* infix, char* postfix) {
Stack stack; // Declare a stack object for operator manipulation
 stack.top = NULL; // Initialize stack to be empty (top pointer points to NULL)
 int i = 0, j = 0; // Indexes for infix and postfix strings
 while (infix[i] != '\0') { // Loop until null terminator in infix string
if (isdigit(infix[i])) { // Check if current character is a digit
 postfix[j++] = infix[i++]; // Copy digit to postfix and increment both indexes
} else if (infix[i] == '(') {
push(&stack, infix[i++]); // Push '(' onto the stack and increment index
} else if (infix[i] == ')') {
 while (!is_empty(&stack) && peek(&stack) != '(') {
 // Pop operators from stack until encountering '(' and append them to postfix
postfix[j++] = pop(&stack);
}
 pop(&stack); // Discard '(' from stack
 i++; // Increment index after processing closing parenthesis
} else if (is_operator(infix[i])) {
 while (!is_empty(&stack) && precedence(peek(&stack)) >=
precedence(infix[i])) {
// Pop operators with higher or equal precedence from stack and append to
postfix
postfix[j++] = pop(&stack);
}
```

```
push(&stack, infix[i++]); // Push current operator onto the stack and increment index
} else {
 // Ignore other characters like spaces (optional handling)
i++;
    }
 } // Pop remaining operators from stack after processing infix string
while (!is_empty(&stack)) {
postfix[j++] = pop(&stack);
}
postfix[j] = '\0'; // Add null terminator to mark the end of the postfix string
}
 // Function to evaluate postfix expression
int evaluate_postfix(char* postfix) {
 Stack stack; // Declare a stack object for operand manipulation
stack.top = NULL; // Initialize stack to be empty
int i = 0; // Index for postfix string
 while (postfix[i] != '\0') { // Loop until null terminator in postfix string
if (isdigit(postfix[i])) {
push(&stack, postfix[i] - '0'); // Convert digit character to integer value and push onto stack
 } else if (is_operator(postfix[i])) {
int operand2 = pop(&stack); // Pop the second operand from the stack
 int operand1 = pop(&stack); // Pop the first operand from the stack
switch (postfix[i]) {
 case '+':
```

```
 push(&stack, operand1 + operand2); // Perform addition and push result onto stack

break;

 case '-':

 push(&stack, operand1 - operand2); // Perform subtraction and push result onto stack

break;

 case '*':

 push(&stack, operand1 * operand2); // Perform multiplication and push result onto stack

 break;

case '/':

 push(&stack, operand1 / operand2); // Perform division and push result onto stack

break;

   }

 }

i++; // Increment index after processing the current character

 }

 return pop(&stack); // The final element on the stack is the result of the expression

}
```

EXPLANATION:

This code provides functions to convert an infix expression to a postfix expression and to evaluate the postfix expression. Here's an explanation of each function:

1. **Function `is_operator`**: This function checks if a given character is one of the common arithmetic operators (`+`, `-`, `*`, `/`) and returns `1` if it is, otherwise `0`.

N. Sai Kiran
03/05/2024

2. **Function `precedence`**: This function assigns precedence levels to operators. It returns `1` for addition and subtraction, `2` for multiplication and division, and `0` for parentheses (which are given the highest precedence).

3. **Function `infix_to_postfix`**: This function converts an infix expression to a postfix expression. It uses a stack to temporarily store operators. It iterates through the infix expression character by character:

- If the character is a digit, it is directly added to the postfix expression.

- If it is an opening parenthesis `'('`, it is pushed onto the stack.

- If it is a closing parenthesis `')'`, operators are popped from the stack and added to the postfix expression until an opening parenthesis is encountered. The opening parenthesis is then popped from the stack.

- If it is an operator, operators with higher or equal precedence are popped from the stack and added to the postfix expression. The current operator is then pushed onto the stack.

- Other characters (like spaces) are ignored.

4. **Function `evaluate_postfix`**: This function evaluates a postfix expression and returns the result. It iterates through the postfix expression character by character:

- If the character is a digit, it is converted to an integer and pushed onto the stack.

- If it is an operator, the required number of operands are popped from the stack, the operation is performed, and the result is pushed back onto the stack.

- After processing the entire expression, the final result is popped from the stack and returned.

These functions work together to convert infix expressions to postfix expressions and evaluate them using stacks.

Expression_evaluator.h:

#ifndef EXPRESSION_EVALUATOR_H // Check if EXPRESSION_EVALUATOR_H hasn't been included yet #define EXPRESSION_EVALUATOR_H // Define header guard to prevent multiple inclusions

// Function prototypes

N. Sai Kiran
03/05/2024

void infix_to_postfix(char* infix, char* postfix); // Prototype for infix to postfix conversion function int evaluate_postfix(char* postfix); // Prototype for postfix expression evaluation function

#endif /* EXPRESSION_EVALUATOR_H */

EXPLANATION:

This header file `EXPRESSION_EVALUATOR_H` contains function prototypes for the infix to postfix conversion function (`infix_to_postfix`) and the postfix expression evaluation function (`evaluate_postfix`). It includes a header guard to prevent multiple inclusions.

By including this header file in other source files that need to use these functions, you can ensure that the functions are declared and available for use without needing to include their implementations. This promotes modularity and code organization.

Main.c:

#include "expression_evaluator.h" // Include header for expression evaluation functions (assumed) #include<stdio.h> // Include standard input/output header

#include <time.h>// Include time header for clock function

#define MAX_EXPR_SIZE 100 //Define Maximum expression size(constant)

int main() {

char infix[MAX_EXPR_SIZE]; // Declare character array to store infix expression

char postfix[MAX_EXPR_SIZE]; // Declare character array to store postfix expression

double start_time, end_time; // Declare variables to store start and end

// Get start time before expression evaluation

start_time = clock(); // Get starting CPU time in clock ticks

printf("Enter an infix expression: ");

fgets(infix, MAX_EXPR_SIZE, stdin); // Read infix expression from user input

infix_to_postfix(infix, postfix); // Convert infix to postfix expression (assuming function exists) printf("Postfix expression: %s\n", postfix); // Print converted postfix expression

int result = evaluate_postfix(postfix); //Evaluate postfix expression(assuming function exists)

printf("Result: %d\n", result); // Print evaluation result

// Get end time after expression evaluation

end_time = clock();   // Get ending CPU time in clock ticks

// Calculate elapsed time in seconds

double elapsed_time = (end_time - start_time) ;/ CLOCKS_PER_SEC

printf("Elapsed time: %.6f seconds\n", elapsed_time);

// Print elapsed time with 6 decimal places

return 0;

}

EXPLANATION:

This `main` function demonstrates the usage of the infix to postfix conversion and postfix expression evaluation functions. Here's how it works:

1. **Includes**: The code includes necessary header files such as `"expression_evaluator.h"` for the function prototypes and standard library headers for input/output (``) and time measurement (``).
2. **Macros**: It defines a macro `MAX_EXPR_SIZE` to specify the maximum size of the input infix and postfix expressions.
3. **Variables**: It declares variables to store the infix and postfix expressions (`infix` and `postfix`), as well as variables to store the start and end times for measuring the execution time of the expression evaluation.

4. **Input**: It prompts the user to enter an infix expression, reads it from the standard input using `fgets`, and stores it in the `infix` array.
5. **Conversion and Evaluation**: It calls the `infix_to_postfix` function to convert the infix expression to postfix and stores the result in the `postfix` array. Then, it calls the `evaluate_postfix` function to evaluate the postfix expression and stores the result in the variable `result`.
6. . **Output**: It prints the converted postfix expression and the evaluation result to the standard output.
7. **Time Measurement**: It measures the elapsed time for expression evaluation using the `clock` function from the `` header. It calculates the elapsed time in seconds and prints it to the standard output.

This `main` function provides a simple way to interactively input infix expressions, convert them to postfix, evaluate them, and measure the time taken for the evaluation.