Data Structure

Homework1:

Author: B11002220 CHI-CHUN, LO

Date: October 7, 2023

NTUST ECE

Contents

1 Problem

A common problem for compilers and text editors is determining whether the parentheses in an expression are balanced and properly nested. For example, the expression $A/[D*(B-F)]+S$ contains probably nested pairs, while $(G*[A-B)]/C$ and $(d+e/A+t)$ do not. We are tasked with creating a program that returns true if an expression contains probably nested and balanced parentheses, and false if otherwise. Additionally, for the balanced case, we need to find the positions for each pair of the opening and closing parentheses. On the other hand, for the unbalanced case, we need to identify the position of the first offending

1.1 Definition

According to the given requirements, we can define a function called **check-parentheses(***expression***)** to determine whether the parentheses in an expression are balanced.

- Define the position and name of each bracket and also call the arry as a stack :

```
#define MAX_SIZE 256
typedef struct {
  int pos;
  char word;
} BracketElement;
BracketElement stack[MAX_SIZE] = { 0 };
int stack_index = -1;

```

- Define ths postion of paired bracket:

```
typedef struct {
  int first;
  int last;
} MatchBracket;
MatchBracket match_arr[MAX_SIZE] = { 0 };
int store = 0;

```

1.2 Details

To solve the problem of determining whether parentheses in an expression are balanced and properly nested, we can use a stack data structure. Here's a high-level approach to solve this problem:

1. Scan the expression from left to right.
2. if you find any parenthesis, push it onto the stack.
3. if you find a closing parenthesis, pop two of it from the stack.

4. at the end , if the stack is empty , then the expression will be balanced.

5. otherwise, it will be unbalanced.

Therefore, we can use pseudocode to show clearly.

Algorithm 1: Checking balanced parentheses

```
Input  : expression
Output: True if expression has balanced parentheses, False otherwise
stack ← empty stack;
MatchBracket_arr ← empty arr;
for each character char in expression do
    if char is a parenthesis then
        push char onto stack;
    if Current_Stack and Previous_stack form closed parentheses then
        pop two elements from the stack;
        record them onto MatchBracket_arr;
if stack is not empty then
    return False;
return True;
```

2 Code

```

    #include <stdio.h>
    #include <stdlib.h>
    #include <string.h>
    // Define a stack from BracketElement
    #define MAX_SIZE 256
    typedef struct {
      int pos;
      char word;
    } BracketElement;
    BracketElement stack[MAX_SIZE] = { 0 };
    int stack_index = -1;

    // Define a match arr to store the array
    typedef struct {
      int first;
      int last;
    } MatchBracket;
    MatchBracket match_arr[MAX_SIZE] = { 0 };
    int store = 0;

    // append to the stack
    void stack_append(BracketElement element) {
      stack[++stack_index] = element;
    }
    // get the element from the stack
    BracketElement stack_pop() {
      return stack[stack_index--];
    }
    // whether is empty
    int stack_is_empty() {
      return (stack_index < 0);
    }
    // whether the stack is full
    int stack_is_full() {
      return (stack_index == MAX_SIZE - 1);
    }
    // func:check_parentheses
    // para:
    //   - str: expression
    // return:
    //   - 1: if balanced
    //   - 0: else
    int check_parentheses(char* str) {
      stack_index = -1;
      store = 0;
      MatchBracket match_temp;
      BracketElement temp;
      for (int i = 0; i < strlen(str); i++) {
        temp.pos = i;
        temp.word = str[i];
        if (str[i] == '(' || str[i] == '[' || str[i] == '{' ||
str[i] == ')' || str[i] == ']' || str[i] == '}') {
          stack_append(temp);
        }
```

```
55          // lens of it more than 1 , and form a close  parathese
56          if (stack_index >= 1 && (
57            (stack[stack_index].word == ')' && stack[stack_index -
    1].word == '(') ||
58            (stack[stack_index].word == ']' && stack[stack_index -
    1].word == '[') ||
59            (stack[stack_index].word == '}' && stack[stack_index -
    1].word == '{')
60            )) {
61            // record and pop two of it from the stack
62            temp = stack_pop();
63            match_temp.last = temp.pos;
64            temp = stack_pop();
65            match_temp.first = temp.pos;
66            match_arr[store] = match_temp;
67            store++;
68          }
69        }
70        return stack_is_empty();
71      }
72      // at the end , to fit the result of it gives ,
73      // use qsort to order the matched arr .
74      int compare(const void* a, const void* b) {
75        MatchBracket* bracket1 = (MatchBracket*)a;
76        MatchBracket* bracket2 = (MatchBracket*)b;
77
78        if (bracket1->first < bracket2->first) {
79          return -1;
80        }
81        else if (bracket1->first > bracket2->first) {
82          return 1;
83        }
84        else {
85          return 0;
86        }
87      }
88      int main() {
89        FILE* file;
90        FILE* OUTPUT_FILE;
91        errno_t err1 = fopen_s(&file, "Input.txt", "r");
92        errno_t err2 = fopen_s(&OUTPUT_FILE, "Output.txt", "w");
93
94        if (err1 != 0 || err2!= 0 ) {
95          printf("Cannot open the file\n");
96          return 1;
97        }
98
99        char line[MAX_SIZE+3];// bounded condition
100       while (fgets(line, MAX_SIZE+3, file) != NULL) {
101         if (check_parentheses(line)){
102           fprintf_s(OUTPUT_FILE, "1\n");
103           // quick sort in order to have correct result.
104           qsort(&match_arr, store, sizeof(match_arr[0]), compare)
    ;
105           //write to Output.txt
106           for (int i = 0; i < store ; i++)
107           {
```

```
108                 fprintf_s(OUTPUT_FILE, "%d,%d;",match_arr[i].first ,
    match_arr[i].last);
109               }
110               fprintf_s(OUTPUT_FILE, "\n");
111               printf("\n");
112             }
113             else{
114               fprintf_s(OUTPUT_FILE, "-1\n");
115               fprintf_s(OUTPUT_FILE, "%d\n",stack[0].pos);
116             }
117           }
118           fclose(file);
119           fclose(OUTPUT_FILE);
120           return 0;
121         }
122
```

3 Result

To ensure the proper functioning of the program under various conditions, several test cases are conducted to verify its correctness. Particularly, the fourth and fifth tests focus on boundary conditions, where the input string has a length of 0 and 256 respectively.

Tests:

```
()()()][[]
{[)()}
{}(())

((((.....)))) *256
```

Expected Results:

```
−1
6
−1
0
1
0,1;2,3;4,5;6,9;7,8;
1

1
0,255;.....127,128;
```

Here is the Output:

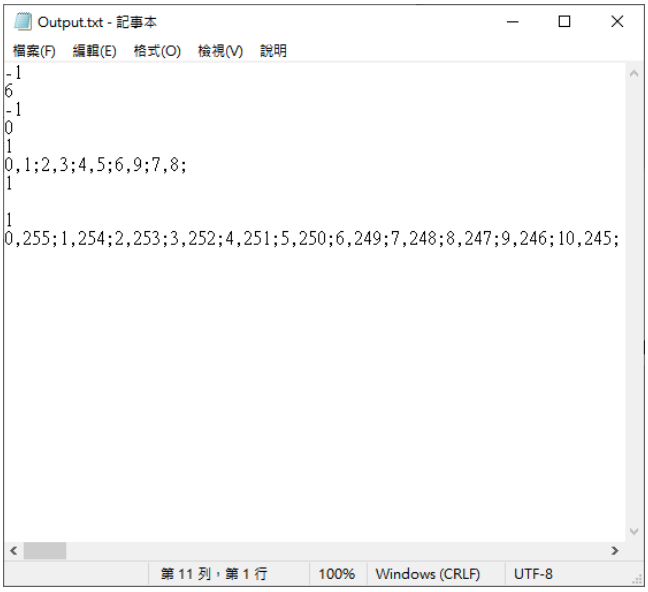

Figure 1: Output Image

4 Discussion and Conclusion

conclusion

I was deeply moved by the test cases I came up with during the final testing phase. Initially, the Program run fourth and fifth test cases were incorrect. However, the issue was often due to a lack of consideration at the beginning, which led to problems in the following code. To solve this problem, it is essential to consider the limitations of input, output, and string-related functions from the very beginning.

Discussion:The Test is very essential

If the string length is 256, the fget function does assign 256 characters to the line variable. However, the range of strlen(line) can only capture 0-255 characters. Therefore, it is necessary to allocate a larger space for the line variable, as shown in the image below.

```
char line[MAX_SIZE];// bounded condition
while (fgets(line, MAX_SIZE, file) != NULL) {
    if (check_parentheses(line)){
```

Figure 2: Original Program

```
char line[MAX_SIZE+2];// bounded condition
while (fgets(line, MAX_SIZE+2, file) != NULL) {
    if (check_parentheses(line)){
        fprintf_s(OUTPUT_FILE, "1\n");
```

Figure 3: Updated Program