CODE:

prog1_4.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "str_func.h"

int main() {
   int chx, argv;
   bool flg = true;
   char *sentence, x, *str2, *str3, *res, **args;

   while(flg) {
       printf("\n");
       printf("1. Remove Letters\n");
       printf("2. Find and Replace\n");
       printf("3. Reverse Sentence\n");
       printf("4. String Formatting\n");
       printf("0. Exit\n");

       printf("\nEnter choice: ");
       scanf("%d", &chx);
       clr_buffr();

       switch(chx) {
           case 0:
               flg = false;
               break;
```

```
case 1:
    sentence = input_str();
    printf("enter char to remove: ");
    scanf("%c", &x);
    clr_buffr();
    res = remove_letter(sentence, x);
    printf("Result: %s\n", res);
    free(sentence);
    free(res);
    break;
case 2:
    printf("Enter the sentence:\n");
    sentence = input_str();
    printf("Enter word to replace:\n");
    str2 = input_str();
    printf("Enter replacement:\n");
    str3 = input_str();
    res = replace_word(sentence, str2, str3);
    printf("Result: %s\n", res);
    free(sentence);
    free(str2);
    free(str3);
    free(res);
    break;
case 3:
    sentence = input_str();
    res = reverse_sentence(sentence);
    printf("Result: %s\n", res);
    free(sentence);
    }

            free(res);
            break;
        case 4:
            printf("Enter the sentence:\n");
            sentence = input_str();
            printf("Enter number of arguments: ");
            scanf("%d", &argv);
            clr_buffr();
            args = malloc(argv*sizeof(char *));
            if (args == NULL) {
                printf("MEMORY ALLOCATION ERROR.\n");
                exit(-1);
            }
            for(int i=0; i<argv; i++) {
                args[i] = input_str();
            }
            res = format_str(sentence, args, argv);
            printf("Result: %s\n", res);

            for(int i=0; i<argv; i++) {
                free(args[i]);
            }
            free(args);
            free(sentence);
            free(res);
            break;
        default:
            printf("Invalid Input.\n");
    }
```

```
   }
   return 0;
}
```

str_func.h

```
#ifndef STR_FUNC_H
#define STR_FUNC_H

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

enum bool {
   false = 0,
   true = 1
};
typedef enum bool bool;


char *format_str(char *description, char **values, int argv);
void reverse_sequence(char *seq, int begin_indx, int
end_indx);
char *reverse_sentence(char *sentence);
int *calc_lps(char *pattern);
int *kmp_search(char *text, char *pattern, int
*no_of_matches);
char *replace_word(char *sentence, char *target, char
*replacement);
```

```c
bool find_in_arr(int *arr, int arr_size, int target);
char *remove_letter(char *sentence, char letter);
char *input_str();
void clr_buffr();

#endif
```

str_func.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "str_func.h"

char *format_str(char *description, char **values, int argv) {
   int len, i, indx, new_len=0, prev_len, word_size;
   char *res = NULL, *end = NULL, *word;

   len = strlen(description);

   indx = 0;
   i = 0;
   while (i<len) {
       // printf("i: %d\n", i);
       if (description[i] == '$') {
           i++;
           indx = strtol(&description[i], &end, 10);
           // printf("Indx: %d, ", indx);
           // check out of bounds
```

```
            if (indx >= argv || indx < 0) {
                word = "ERROR";
            } else {
                word = values[indx];
            }
            prev_len = new_len;
            word_size = strlen(word);
            new_len += word_size;

            res = realloc(res, new_len*sizeof(char));
            if (res == NULL) {
                printf("MEMORY ALLOCATION ERROR.\n");
                exit(-1);
            }

            for (int k=0; k<word_size; k++) {
                res[prev_len+k] = word[k];
            }
            // incr i
            // while (&description[i] != end) i++;
            i = (int)((int)(end - description) / sizeof(char));
        } else {
            new_len++;
            res = realloc(res, new_len*sizeof(char));
            if (res == NULL) {
                printf("MEMORY ALLOCATION ERROR.\n");
                exit(-1);
            }
            res[new_len - 1] = description[i];
```

```
           i++;
       }
   }
   return res;
}

void reverse_sequence(char *seq, int begin_indx, int end_indx)
{
   int i, j;
   char tmp;
   i = begin_indx;
   j = end_indx;

   while(i < j) {
       tmp = seq[i];
       seq[i] = seq[j];
       seq[j] = tmp;
       i++;
       j--;
   }
}

char *reverse_sentence(char *sentence) {
   char *res = NULL;
   int i, j;
   int len = strlen(sentence);

   res = malloc((len+1)*sizeof(char)); // +1 for '\0'
   if (res == NULL) {
       printf("MEMORY ALLOCATION ERROR.\n");
       exit(-1);
   }

   strcpy(res, sentence);
   i = 0;
   j = 0;

   while(j<(len+1)) {
       if(res[j] == ' ' || res[j] == '\0' || res[j] == '\n') {
           reverse_sequence(res, i, j-1);
           j++;
           i = j;
       } else {
           j++;
       }
   }

   reverse_sequence(res, 0, len-1); // -1 to not reverse the
'\0'

   return res;
}

int *calc_lps(char *pattern) {
   int *lps = NULL, m, curr_len, i;

   m = strlen(pattern);
```

```
    lps = calloc(m, sizeof(int)); // calloc will initialise to 0
    if (lps == NULL) {
        printf("MEMORY ALLOCATION ERROR.\n");
        exit(-1);
    }
    curr_len = 0;
    i = 1;

    while (i < m) {
        if (pattern[curr_len] == pattern[i]) {
            curr_len++;
            lps[i] = curr_len;
            i++;
        } else {
            if (curr_len > 0) {
                // i.e. this will keep executing (due to outer loop) till curr_len becomes 0
                curr_len = lps[curr_len - 1];
            }
            else {
                // i.e. when curr_len has become 0
                lps[i] = 0;
                i++;
            }
        }
    }

    return lps;
```

```
}

int *kmp_search(char *text, char *pattern, int *no_of_matches)
{
   int m, n, i, j, *lps = NULL, *result = NULL;
   n = strlen(text);
   m = strlen(pattern);
   lps = calc_lps(pattern);
   *no_of_matches = 0;
   i = j = 0;

   while (i < n){
       if (pattern[j] == text[i]){
           i++;
           j++;
       }
       if (j == m) {
           (*no_of_matches)++;
           result = realloc(result,
(*no_of_matches)*sizeof(int));
           if (result == NULL) {
               printf("MEMORY ALLOCATION ERROR.\n");
               exit(-1);
           }
           result[(*no_of_matches)-1] = i - j;
           j = lps[j - 1];
       }
       else if (i < n && pattern[j] != text[i]) {
           if (j != 0)
```

```
               j = lps[j - 1];
           else
               i++;
       }
   }
   free(lps);

   return result;
}

char *replace_word(char *sentence, char *target, char
*replacement) {
   int *indices, no_of_matches, old_len, new_len=0, i, prev;
   char *res = NULL;

   old_len = strlen(sentence);
   indices = kmp_search(sentence, target, &no_of_matches);

   i = 0;
   while (i<old_len+1) { //+1 for '\0'
       if (find_in_arr(indices, no_of_matches, i)) {
           // replace
           prev = new_len;
           new_len += strlen(replacement);
           res = realloc(res, new_len*sizeof(char));
           if (res == NULL) {
               printf("MEMORY ALLOCATION ERROR.\n");
               exit(-1);
           }
```

```
            for (int k=prev; k<new_len; k++) {
                res[k] = replacement[k-prev];
            }
            // incr i appropraitely
            i += strlen(target);
        } else {
            new_len++;
            res = realloc(res, new_len*sizeof(char));
            if (res == NULL) {
                printf("MEMORY ALLOCATION ERROR.\n");
                exit(-1);
            }
            res[new_len-1] = sentence[i];
            i++;
        }
    }
    free(indices);

    return res;
}

bool find_in_arr(int *arr, int arr_size, int target) {
    bool found = false;

    for (int i=0; i<arr_size; i++) {
        if (arr[i] == target) {
            found = true;
            break;
        }
    }

    return found;
}

char *remove_letter(char *sentence, char letter) {
    char *res = NULL;
    int old_len, new_len = 0;

    old_len = strlen(sentence);

    for (int i=0; i<old_len+1; i++) { // old_len+1 to scan and add '\0' at end
        if (sentence[i] == letter) {
            continue;
        } else {
            new_len++;
            res = realloc(res, new_len*sizeof(char));
            if (res == NULL) {
                printf("MEMORY ALLOCATION ERROR.\n");
                exit(-1);
            }
            res[new_len-1] = sentence[i];
        }
    }

    return res;
}
```

```c
char *input_str() {
    char *res = NULL, x;
    int size = 0;

    printf("Enter the string: ");
    do {
        x = getchar();
        size++;
        res = realloc(res, size*sizeof(char));
        if (res == NULL) {
            printf("MEMORY ALLOCATION ERROR.\n");
            exit(-1);
        }

        if (x != '\n') {
            res[size-1] = x;
        } else {
            res[size-1] = '\0';
        }
    } while(x != '\n');

    return res;
}

void clr_buffr() {
    char x;
    do {
        x = getchar();
```

```
   } while(x != '\n');
}
```

prog5.c

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

struct tree_node {
   char data;  // Token or operator
   struct tree_node *left;
   struct tree_node *right;
};
typedef struct tree_node tree_node;

// Function to create a new node
tree_node *create_node(char data) {
   struct tree_node *new_node = (struct
tree_node*)malloc(sizeof(struct tree_node));
   new_node->data = data;
   new_node->left = NULL;
   new_node->right = NULL;
   return new_node;
}

// Function to build the parse tree
```

```
tree_node *build_parse_tree(char *expression, int start, int
end) {
   if (start > end) {
       return NULL;
   }

   tree_node *root = create_node(expression[start]);

   // Check for operators
   if (start + 1 <= end && expression[start + 1] == '/' ||
expression[start + 1] == '*' ||
       expression[start + 1] == '+' || expression[start + 1]
== '-') {

       root->left = create_node(expression[start + 1]);
       root->right = build_parse_tree(expression, start + 2,
end);
   } else {
       // Operand case
       root->left = build_parse_tree(expression, start + 1,
end);
   }

   return root;
}

// Function to print the parse tree in an inorder traversal
void print_parse_tree_in(tree_node *root) {
   if (root != NULL) {
       print_parse_tree_in(root->left);
       printf("%c, ", root->data);
       print_parse_tree_in(root->right);
   }
}

// Function to print the parse tree in an preorder traversal
void print_parse_tree_pre(tree_node *root) {
   if (root != NULL) {
       printf("%c, ", root->data);
       print_parse_tree_pre(root->left);
       print_parse_tree_pre(root->right);
   }
}

bool is_operator(char x) {
   return (x == '*' || x == '/' || x == '+' || x == '-' || x
== '=');
}

char **lexicial_analyser(char *expression, int *no_of_op, int
*no_of_id) {
   char *operators = NULL, *identifiers = NULL, **res = NULL;
   int len = strlen(expression);

   for (int i=0; i<len; i++) {
       if(is_operator(expression[i])) {
           // append to operators array
           (*no_of_op)++;
```

```
            operators = realloc(operators,
(*no_of_op)*(sizeof(char)));
            if (operators == NULL) {
                printf("MEMORY ALLOCATION ERROR.\n");
                exit(-1);
            }
            operators[(*no_of_op) - 1] = expression[i];
        } else {
            if (expression[i] == ' ') {
                continue;
            }
            (*no_of_id)++;
            identifiers = realloc(identifiers,
(*no_of_id)*(sizeof(char)));
            if (identifiers == NULL) {
                printf("MEMORY ALLOCATION ERROR.\n");
                exit(-1);
            }
            identifiers[(*no_of_id) - 1] = expression[i];
        }
    }
    res = malloc(2*sizeof(char *));
    if (res == NULL) {
        printf("MEMORY ALLOCATION ERROR.\n");
        exit(-1);
    }
    res[0] = operators;
    res[1] = identifiers;
```

```c
    return res;
}

char *trim_spaces(char *expression) {
    int len, new_len = 0;
    char *res = NULL;
    len = strlen(expression);

    for (int i=0; i<len+1; i++) {
        if (expression[i] == ' ') {
            continue;
        }

        new_len++;
        res = realloc(res, new_len*sizeof(char));
        if (res == NULL) {
            printf("MEMORY ALLOCATION ERROR.\n");
            exit(-1);
        }
        res[new_len - 1] = expression[i];
    }
    return res;
}

int search_symbols(char query) {
    int indx = -1;
    bool found = false;
    char symbols[10] = {'0', '1', '2', '3', '4', '5', '6', '7',
'8', '9'};
```

```
    for (int i=0; i<10; i++) {
        if (symbols[i] == query) {
            indx = i;
            found = true;
            break;
        }
    }

    if (!found) {
        indx = -1;
    }

    return indx;
}

void semantic_analyser(char **res, int no_of_id, int no_of_op)
{
    int idf;

    // printf("operator: ");
    for (int i=0; i<no_of_id; i++) {
        idf = search_symbols(res[1][i]);

        if (idf == -1) {
            printf("Undefined Symbol: %c\n", res[1][i]);
        } else {
            printf("Int Symbol Detected: %c\n", res[1][i]);
        }
    }
}


int main() {
    char *expression = input_str();
    // int length = sizeof(expression) / sizeof(expression[0]);
    int no_of_op, no_of_id;
    char **res;

    char *exp = trim_spaces(expression);

    res = lexicial_analyser(exp, &no_of_op, &no_of_id);
    printf("Output of Lexical Analysis:\n");

    printf("operator: ");
    for (int i=0; i<no_of_op; i++) {
        printf("%c, ", res[0][i]);
    }
    printf("\n");
    printf("ids: ");
    for (int i=0; i<no_of_id; i++) {
        printf("%c, ", res[1][i]);
    }
    printf("\n");

    tree_node *root = build_parse_tree(exp, 0, strlen(exp) -
1);
    printf("\nOutput of Syntactic Analysis:\n");
```

```c
   printf("Parse Tree (Inorder Traversal):\n");
   print_parse_tree_in(root);
   printf("\n");
   printf("Parse Tree (Preorder Traversal):\n");
   print_parse_tree_pre(root);
   printf("\n");

   printf("\nOutput of Semantic Analysis:\n");
   semantic_analyser(res, no_of_id, no_of_op);

   return 0;
}
```

OUTPUTS:

prog1_4.c

```
1. Remove Letters
2. Find and Replace
3. Reverse Sentence
4. String Formatting
0. Exit

Enter choice: 1
Enter the string: Hello there
enter char to remove: e
Result: Hllo thr

1. Remove Letters
2. Find and Replace
3. Reverse Sentence
4. String Formatting
0. Exit
```

```
Enter choice: 2
Enter the sentence:
Enter the string: Hello guys, Hello everyone
Enter word to replace:
Enter the string: Hello
Enter replacement:
Enter the string: Hi
Result: Hi guys, Hi everyone

1. Remove Letters
2. Find and Replace
3. Reverse Sentence
4. String Formatting
0. Exit

Enter choice: 3
Enter the string: Good morning everyone
Result: everyone morning Good

1. Remove Letters
2. Find and Replace
3. Reverse Sentence
4. String Formatting
0. Exit


1. Remove Letters
2. Find and Replace
3. Reverse Sentence
4. String Formatting
0. Exit

Enter choice: 4
Enter the sentence:
Enter the string: Hi my $0, $1 and $2.
Enter number of arguments: 2
Enter the string: friends
Enter the string: Ramesh
```

```
Result: Hi my friends, Ramesh and ERROR.
```

prog5.c

```
Enter the string: a /b * c - d + e
Output of Lexical Analysis:
operator: /, *, -, +,
ids: a, b, c, d, e,

Output of Syntactic Analysis:
Parse Tree (Inorder Traversal):
/, a, *, b, -, c, +, d, e,
Parse Tree (Preorder Traversal):
a, /, b, *, c, -, d, +, e,

Output of Semantic Analysis:
Undefined Symbol: a
Undefined Symbol: b
Undefined Symbol: c
Undefined Symbol: d
Undefined Symbol: e


Enter the string: 2 + 3 - x
Output of Lexical Analysis:
operator: +, -,
ids: 2, 3, x,

Output of Syntactic Analysis:
Parse Tree (Inorder Traversal):
+, 2, -, 3, x,
Parse Tree (Preorder Traversal):
2, +, 3, -, x,

Output of Semantic Analysis:
Int Symbol Detected: 2
Int Symbol Detected: 3
Undefined Symbol: x
```