Experiment No.4
Design & Implementation of Pass 2 of Two Pass Macro Processor
Date of Performance: 17-02-2025
Date of Submission: 24-02-2025

Aim: Design & Implementation of Pass 2 of Two Pass Macro Processor.

Objective: To study and implement Pass 2 of two pass Macro Processor for IBM 360 Machine.

Theory:

Macro: A macro is a unit of specification for program generation through expansion. A macro instruction is a notational convenience for the programmer. It allows the programmer to write shorthand version of a program (module programming).

Macro Processor: The macro processor replaces each macro invocation with the corresponding sequence of statements.

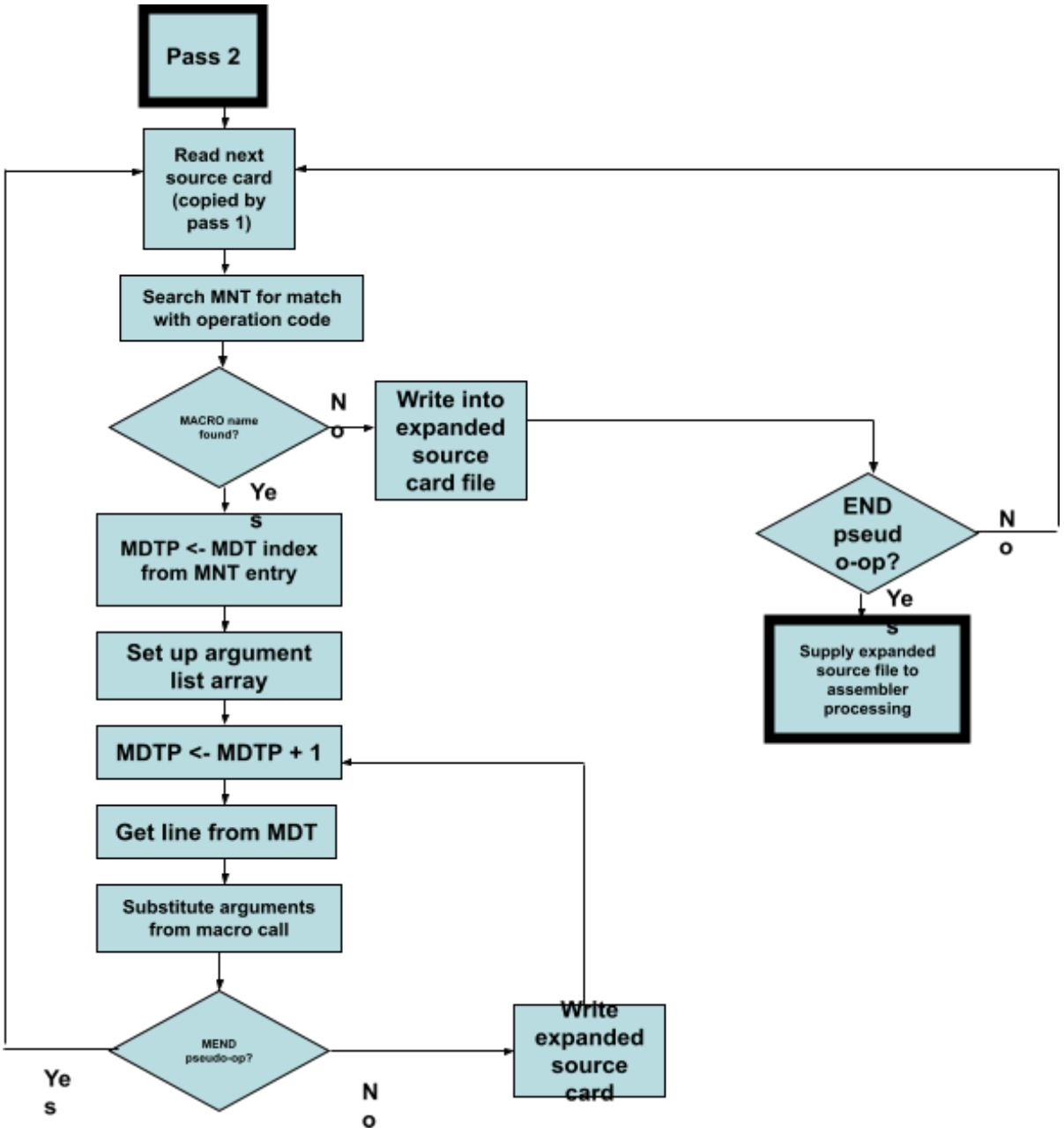
Pass 2: Expand all macro invocation statements.

- Monitors the values of expansion time variables & sequencing symbols specified in a Macro
- Handles expansion time control flow & performs expansion of model statements.

Pass 2 Databases of Macro Processor

1. Copy of source program statements
2. Output expanded source listings for input to assembler
3. MNT and MDT generated in pass 1
4. Macro Definition Table Pointer (MDTP) directs to the next statement to be used during expansion
5. An array called Parameter List Array (PLA) for substituting parameters of macro call in the source macro of the macro call in the stored macro definitions for the index correspondingly.

Pass 2 flowchart:



Code:

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

#define MAX_MACRO 10
#define MAX_MDT 100
#define MAX_ALA 10
#define MAX_LINES 100
#define MAX_LENGTH 100

typedef struct {
    char name[20];
    int index;
} MNTEntry;

typedef struct {
    char opcode[20];
    char operand[50];
} MDTEntry;

typedef struct {
    char formal[20];
    char actual[20];
```

```
} ALAEntry;


MNTEntry mnt[MAX_MACRO];
MDTEntry mdt[MAX_MDT];
ALAEntry ala[MAX_ALA];
int mntc = 0, mdtc = 0, alac = 0;


void add_to_ala(char *param) {
    sprintf(ala[alac].formal, "%s", param);
    sprintf(ala[alac].actual, "#%d", alac);
    alac++;
}


void process_macro_definition(char lines[MAX_LINES][MAX_LENGTH], int start, int end)
{
    char macro_name[20], params[50];
    char *token;
    sscanf(lines[start], "MACRO %s %49[^"]", macro_name, params);


    strcpy(mnt[mntc].name, macro_name);
    mnt[mntc].index = mdtc;
    mntc++;
```

```
alac = 0;
token = strtok(params, ", ");
while (token) {
        add_to_ala(token);
        token = strtok(NULL, ", ");
}

strcpy(mdt[mdtc].opcode, macro_name);
strcpy(mdt[mdtc].operand, params);
mdtc++;

for (int i = start + 1; i < end; i++) {
        char opcode[20], operand[50], final_operand[50] = "";
        sscanf(lines[i], "%s %[^"]", opcode, operand);

        token = strtok(operand, ", ");
        while (token) {
        int found = 0;
        for (int j = 0; j < alac; j++) {
                if (strcmp(token, ala[j].formal) == 0) {
                strcat(final_operand, ala[j].actual);
                found = 1;
                break;
```

```
                }
        }
        if (!found) strcat(final_operand, token);
        token = strtok(NULL, ", ");
        if (token) strcat(final_operand, ", ");
        }

        strcpy(mdt[mdtc].opcode, opcode);
        strcpy(mdt[mdtc].operand, final_operand);
        mdtc++;
    }

    strcpy(mdt[mdtc].opcode, "MEND");
    strcpy(mdt[mdtc].operand, "");
    mdtc++;
}

void expand_macro(char *macro_name, char *actual_params) {
    int index = -1;
    for (int i = 0; i < mntc; i++) {
        if (strcmp(mnt[i].name, macro_name) == 0) {
        index = mnt[i].index;
        break;
```

```
        }
    }
    if (index == -1) return;

    char actual_values[MAX_ALA][20];
    int actual_count = 0;
    char *token = strtok(actual_params, ", ");
    while (token) {
        strcpy(actual_values[actual_count++], token);
        token = strtok(NULL, ", ");
    }

    for (int i = index + 1; i < mdtc; i++) {
        if (strcmp(mdt[i].opcode, "MEND") == 0) break;

        printf("%s ", mdt[i].opcode);
        char operand_copy[50];
        strcpy(operand_copy, mdt[i].operand);
        token = strtok(operand_copy, ", ");
        while (token) {
        if (token[0] == '#') {
                int param_index = token[1] - '0';
                printf("%s", actual_values[param_index]);
```

```
        } else {
                printf("%s", token);
        }
        token = strtok(NULL, ", ");
        if (token) printf(", ");
        }
        printf("\n");
    }
}

void process_source_code(char lines[MAX_LINES][MAX_LENGTH], int total_lines) {
    for (int i = 5; i < total_lines; i++) {
        char opcode[20], operand[50];
        sscanf(lines[i], "%s %[^"]", opcode, operand);

        int is_macro_call = 0;
        for (int j = 0; j < mntc; j++) {
        if (strcmp(mnt[j].name, opcode) == 0) {
                is_macro_call = 1;
                expand_macro(opcode, operand);
                break;
        }
        }
```

```
            if (!is_macro_call) printf("%s %s\n", opcode, operand);
        }
    }

int main() {
    FILE *file = fopen("input.asm", "r");
    if (!file) {
            printf("Error: Could not open input.asm\n");
            return 1;
    }

    char lines[MAX_LINES][MAX_LENGTH];
    int total_lines = 0;
    while (fgets(lines[total_lines], MAX_LENGTH, file) && total_lines < MAX_LINES) {
        lines[total_lines][strcspn(lines[total_lines], "\n")] = '\0';
            total_lines++;
    }
    fclose(file);

    int macro_start = -1;
    for (int i = 0; i < total_lines; i++) {
            if (strncmp(lines[i], "MACRO", 5) == 0) {
            macro_start = i;
```

```
        } else if (strncmp(lines[i], "MEND", 4) == 0 && macro_start != -1) {
        process_macro_definition(lines, macro_start, i + 1);
        macro_start = -1;
        }
    }

    process_source_code(lines, total_lines);
    return 0;
}
```

Output:

```
Macro Name Table (MNT):
--------------------------------
Index   Macro Name   MDT Index
--------------------------------
0       INCR        0

Argument List Array (ALA):
----------------------------
Index   Formal    Actual
----------------------------
#0      X        N1
#1      Y        N2
#2      Z        AREG

Macro Definition Table (MDT):
---------------------------------------
Index   Opcode   Operands
---------------------------------------
0       INCR    &X, &Y, &Z
1       MOVER   #2, #0
2       MOVER   #2, #1
3       MOVER   #2, #0
4       MEND
```

```
Final Expanded Code (Pass 2 Output):
START 100
READ N1
READ N2
MOVER AREG, N1
MOVER AREG, N2
MOVER AREG, N1
STOP
```

Conclusion:

→ 1. Working of Program by Providing Input Source Program :

Pass 2 of the Two Pass Macro Processor focuses on the expansion of macro invocations using the tables generated in Pass 1. When an input source program containing macro calls is provided, the processor retrieves the macro definition from the Macro Definition Table (MDT), maps actual arguments using the Argument List Array (ALA), and replaces formal parameters with actual values. It then generates the expanded code line by line. This process ensures accurate macro expansion and seamless integration into the final source program. The program demonstrates efficient working by correctly substituting macro bodies wherever invoked in the input code.

→ 2. Output Generated During Processing Macro Invocation :

During the processing of macro invocations in Pass 2, the output consists of fully expanded source code where all macro calls are replaced by their corresponding instruction sequences from the Macro Definition Table (MDT). The actual arguments are accurately substituted for formal parameters using the Argument List Array (ALA), ensuring functional correctness. This expanded code is ready for the next stages of assembly or compilation without requiring macro-related references. The generated output is clear, structured, and maintains logical consistency with the original program intent, highlighting the effectiveness of macro processing and automation provided by the Two Pass Macro Processor.