BISON-ASSIGNMENT 10

Write a program using *bison* that creates a sequence of simple steps (three-address code) according to a given grammar for arithmetic expressions. For example, if the input is-   a + b * c + d

the program should print the output as-

```
    T1 = b * c
    T2 = a + T1
    T3 = T2 + d
```

[Hint: use a global variable, say *t* initialised to 1. Store its value as an attribute of the LHS grammar symbol whenever a reduce-action is performed for a simple arithmetic operation., and increment *t* (so that for the next use it has a different value). Use this attribute value whenever that symbol appears on the RHS for a subsequent reduce action. When printing print this value along with the letter T.]

Expression.l

```
%{
#include "expression.tab.h"
%}

%%
[a-zA-Z][a-zA-Z0-9]* { yylval = strdup(yytext); return ID; }
[0-9]+               { yylval = strdup(yytext); return NUMBER; }
[ \t\n]              ; /* skip whitespace */
.                    { return yytext[0]; }
%%

int yywrap() {
    return 1;
}
```

Bison.y

```
%{
#include <stdio.h>

int t = 1; // Global variable t for generating temporary variables
%}

%token ID
%token NUMBER
%left '+' '-'
%left '*' '/'

%%

stmt: expr { printf("%s = %s\n", $$.name, $1.name); } ;

expr: expr '+' expr {
    $$.name = malloc(sizeof(char) * 10); // Allocate memory for temporary
variable name
    sprintf($$.name, "T%d", t++);
    printf("%s = %s + %s\n", $$.name, $1.name, $3.name);
}
| expr '-' expr {
    $$.name = malloc(sizeof(char) * 10);
    sprintf($$.name, "T%d", t++);
    printf("%s = %s - %s\n", $$.name, $1.name, $3.name);
}
| expr '*' expr {
    $$.name = malloc(sizeof(char) * 10);
    sprintf($$.name, "T%d", t++);
    printf("%s = %s * %s\n", $$.name, $1.name, $3.name);
}
| expr '/' expr {
    $$.name = malloc(sizeof(char) * 10);
    sprintf($$.name, "T%d", t++);
    printf("%s = %s / %s\n", $$.name, $1.name, $3.name);
}
| '(' expr ')' {
    $$.name = $2.name;
}
| ID {
    $$.name = strdup($1); // Duplicate the ID token value
}
| NUMBER {
    $$.name = strdup($1); // Duplicate the NUMBER token value
}
;

%%

int main() {
    yyparse();
    return 0;
}

int yyerror(char* s) {
    printf("Error: %s\n", s);
    return 0;
}
```

Leading and trailing ASSIGNMENT 9

Incorporate methods to automatically create the LEADING and TRAILING sets for a context free grammar, in the program written for the most recent operator precedence parsing assignment.

Lead_trail.c

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

```
#include <stdlib.h>

struct production {
  char *left_hand_side;
  char **right_hand_side;
};

struct grammar {
  struct production *productions;
  int num_productions;
};

struct leading_set {
  char *symbol;
  struct leading_set *next;
};

struct trailing_set {
  char *symbol;
  struct trailing_set *next;
};

struct leading_set *create_leading_set(char *symbol) {
  struct leading_set *set = malloc(sizeof(struct leading_set));
  set->symbol = symbol;
  set->next = NULL;
  return set;
}

struct trailing_set *create_trailing_set(char *symbol) {
  struct trailing_set *set = malloc(sizeof(struct trailing_set));
  set->symbol = symbol;
  set->next = NULL;
  return set;
}

void add_to_leading_set(struct leading_set **set, char *symbol) {
  struct leading_set *new_set = create_leading_set(symbol);
  new_set->next = *set;
  *set = new_set;
}

void add_to_trailing_set(struct trailing_set **set, char *symbol) {
  struct trailing_set *new_set = create_trailing_set(symbol);
  new_set->next = *set;
  *set = new_set;
}

void create_leading_and_trailing_sets(struct grammar *grammar) {
  struct leading_set *leading_sets = NULL;
  struct trailing_set *trailing_sets = NULL;

  for (int i = 0; i < grammar->num_productions; i++) {
    add_to_leading_set(&leading_sets, grammar->productions[i].left_hand_side);
    for (int j = 0; grammar->productions[i].right_hand_side[j] != NULL; j++) {
      add_to_trailing_set(&trailing_sets, grammar->productions[i].right_hand_side[j]);
    }
  }

  grammar->leading_sets = leading_sets;
  grammar->trailing_sets = trailing_sets;
}

int main() {
  struct grammar grammar = {
    .productions = (struct production []) {
      { "E", (char *[]) { "E", "+", "T" } },
      { "E", (char *[]) { "E", "-", "T" } },
      { "E", (char *[]) { "T" } },
      { "T", (char *[]) { "T", "*", "F" } },
      { "T", (char *[]) { "T", "/", "F" } },
      { "T", (char *[]) { "F" } },
      { "F", (char *[]) { "(", "E", ")" } },
      { "F", (char *[]) { "id" } },
      { "F", (char *[]) { "number" } },
    },
    .num_productions = 8,
  };

  create_leading_and_trailing_sets(&grammar);

  for (struct leading_set *set = grammar.leading_sets; set != NULL; set = set->next) {
    printf("LEADING: %s\n", set->symbol);
  }

  for (struct trailing_set *set = grammar.trailing_sets; set != NULL; set = set->next) {
    printf("TRAILING: %s\n", set->symbol);
  }
  return 0;
}
```

Creating an Operator Precedence Parsing Table

ASSIGNMENT 8

Imagine the syntax of a programming language construct such as *while-loop* -- while (*condition*)

```
  begin
     statement ;
           :
  end
```

where *while, begin, end* are keywords; *condition* can be a single comparision expression (such as *x == 20*, etc.); and *statement* is the assignment to a location the result of a single arithmatic operation (eg., *a = 10 * b*).

Represent the above construct in a context free grammar , and write a program to create the operator precedence parsing table. Implement simplified LEADING and TRAILING sets for the non-terminals of the grammar. Hint: First represent the grammar in appropriate data structures.

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

struct production {
  char *left_hand_side;
  char **right_hand_side;
};

struct grammar {
  struct production *productions;
  int num_productions;
};

struct leading_set {
  char *symbol;
  struct leading_set *next;
};

struct trailing_set {
  char *symbol;
  struct trailing_set *next;
};
```

```
struct leading_set *create_leading_set(char *symbol) {
  struct leading_set *set = malloc(sizeof(struct leading_set));
  set->symbol = symbol;
  set->next = NULL;
  return set;
}

struct trailing_set *create_trailing_set(char *symbol) {
  struct trailing_set *set = malloc(sizeof(struct trailing_set));
  set->symbol = symbol;
  set->next = NULL;
  return set;
}

void add_to_leading_set(struct leading_set **set, char *symbol) {
  struct leading_set *new_set = create_leading_set(symbol);
  new_set->next = *set;
  *set = new_set;
}

void add_to_trailing_set(struct trailing_set **set, char *symbol) {
  struct trailing_set *new_set = create_trailing_set(symbol);
  new_set->next = *set;
  *set = new_set;
}

void create_leading_and_trailing_sets(struct grammar *grammar) {
  struct leading_set *leading_sets = NULL;
  struct trailing_set *trailing_sets = NULL;

  for (int i = 0; i < grammar->num_productions; i++) {
    add_to_leading_set(&leading_sets, grammar->productions[i].left_hand_side);
    for (int j = 0; grammar->productions[i].right_hand_side[j] != NULL; j++) {
      add_to_trailing_set(&trailing_sets, grammar->productions[i].right_hand_side[j]);
    }
  }

  grammar->leading_sets = leading_sets;
  grammar->trailing_sets = trailing_sets;
}

int main() {
  struct grammar grammar = {
    .productions = (struct production []) {
      { "while-loop", (char *[]) { "while", "condition", "begin", "statement", "end" } },
      { "condition", (char *[]) { "id", "op", "number" } },
      { "statement", (char *[]) { "id", "op", "number" } },
    },
    .num_productions = 3,
  };

  create_leading_and_trailing_sets(&grammar);

  // Create the operator precedence table.
  struct operator_precedence_table table = {
    .num_operators = 2,
    .operators = (char *[]) { "op", "=" },
    .precedence = (int []) { 1, 2 },
  };

  // Print the operator precedence table.
  for (int i = 0; i < table.num_operators; i++) {
    printf("Operator: %s, Precedence: %d\n", table.operators[i], table.precedence[i]);
  }

  return 0;
}
```

Operator Precedence Parsing

ASSIGNMENT 7

Implement a desk calculator using operator precedence parsing.
[Use flex, and a stack; define a operator precedence table. Push the operands, the operators and precedence relations into the stack till we find that the relationship between the topmost operator on the stack and the next input operator is "takes precedence over" (TPO). When TPO relationship is encountered, compute the topmost operator on the stack along with any other operator below it with "has same precedence as" relationship. Push the result on to the stack as an operand.]

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

#include "flex.h"

#define MAX_STACK_SIZE 100

struct stack_element {
  int type;
  double value;
};

struct stack {
  struct stack_element elements[MAX_STACK_SIZE];
  int top;
};

void init_stack(struct stack *stack) {
  stack->top = 0;
}

void push(struct stack *stack, int type, double value) {
  stack->elements[stack->top].type = type;
  stack->elements[stack->top].value = value;
  stack->top++;
}

double pop(struct stack *stack) {
  double value = stack->elements[stack->top - 1].value;
  stack->top--;
  return value;
}

int get_operator_precedence(char op) {
  switch (op) {
    case '+':
      return 1;
    case '-':
      return 1;
    case '*':
      return 2;
    case '/':
      return 2;
    default:
      return 0;
  }
}

int main() {
  struct stack stack;
  init_stack(&stack);
  // Create the lexer.
  yylex_init();

  // Read the input expression.
  char *expression = malloc(1000);
  fgets(expression, 1000, stdin);

  // Parse the expression.
  int token_type;
  double value;
  while ((token_type = yylex()) != TOK_EOF) {
    switch (token_type) {
      case TOK_NUMBER:
        value = atof(yytext);
        push(&stack, TOKEN_NUMBER, value);
        break;
      case TOK_OPERATOR:
        int precedence = get_operator_precedence(yytext[0]);
        while (stack.top > 0 && get_operator_precedence(stack.elements[stack.top - 1].value) >=
precedence) {
          double operand2 = pop(&stack);
          double operand1 = pop(&stack);
          int operator_type = stack.elements[stack.top - 1].type;
          double result = 0.0;
          switch (operator_type) {
            case TOKEN_ADD:
              result = operand1 + operand2;
              break;
            case TOKEN_SUBTRACT:
              result = operand1 - operand2;
              break;
            case TOKEN_MULTIPLY:
              result = operand1 * operand2;
              break;
            case TOKEN_DIVIDE:
              result = operand1 / operand2;
              break;
          }
          push(&stack, TOKEN_NUMBER, result);
        }
        break;
    }
  }
  // Print the result.
  double result = pop(&stack);
  printf("%.2f\n", result);

  return 0;
}
```

Macro Preprocessing

Assignment-6
Assume that in the assembly used in the previous assignments, there is a statement -DEFINE *name replace-pattern*
The meaning of the statement is that afterwards in the program wherever the pattern *name* occurs, it will be replaced by *replace-pattern*. For example if there is the statement
DEFINE TRUE 1,
then everywhere in the subsequent part in the program if there is the word TRUE, it will be replaced by 1. An input program may contain several such define statements. Write a program to implement this feature.
[This is a very simple macro pre-processor]

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

#define MAX_DEFINES 100

struct define {
  char *name;
  char *replace;
};

struct define defines[MAX_DEFINES];
int num_defines = 0;

void add_define(char *name, char *replace) {
  if (num_defines == MAX_DEFINES) {
    printf("Too many defines\n");
    exit(1);
  }

  defines[num_defines].name = name;
  defines[num_defines].replace = replace;
  num_defines++;
}

void process_line(char *line) {
  char *token;

  token = strtok(line, " ");
  if (token == NULL) {
    return;
  }

  if (strcmp(token, "DEFINE") == 0) {
    token = strtok(NULL, " ");
    if (token == NULL) {
      printf("Missing name for define\n");
      exit(1);
    }

    token = strtok(NULL, " ");
    if (token == NULL) {
      printf("Missing replace pattern for define\n");
      exit(1);
    }

    add_define(token, token);
  } else {
    for (int i = 0; i < num_defines; i++) {
      char *replace = defines[i].replace;
      while (strstr(line, defines[i].name) != NULL) {
        line = strstr(line, defines[i].name);
        int len = strlen(defines[i].name);
        line = strncpy(line, replace, len);
      }
    }

    printf("%s\n", line);
  }
}

int main() {
  char line[1000];

  while (fgets(line, 1000, stdin) != NULL) {
    process_line(line);
  }

  return 0;
}
```

flex

Carry out the Lexical Analyser assignment (given on 6 Feb) using flex utility. At the end of analysing the input assembly language program and printing the output, the lexical analyser should print the total number of lines (i.e., the count) found in the input.

Assignment 5

Token_flex.h

```
#ifndef TOKEN_H
#define TOKEN_H

typedef enum {
    // Imperative statements
    STOP = 1,
    ADD,
    SUB,
    MULT,
    LOAD,
    STORE,
    TRANS,
    TRIM,
    DIV,
    READ,
    PRINT,
    LIR,
    IIR,
    LOOP,

    // Assembler directives
    START,
    END,
    LTORG,

    // Declarative statements
    DS,
    DC,

    // Other tokens
    COMMA,
    LPAREN,
    RPAREN,
    IDENTIFIER,
    NUMBER,
    LABEL
} token_type;
```

```
typedef struct {
    token_type type;
    char *value;
} token;

#endif
```

Lexer_flex.c

```
%{
#include "token.h"
%}

/* Define keywords */
STOP    "00"
ADD     "01"
SUB     "02"
MULT    "03"
TERM    "04"
SUB_NUM "05"
DIV     "06"
READ    "07"
PRINT   "09"
LIR     "10"
IIR     "11"
LOOP    "12"
STORE   "13"
LOAD    "14"
TRANS   "15"
TRIM    "16"

%%

/* Whitespace */
[ \t\n\r]   { /* ignore */ }

/* Mnemonic opcodes */
{STOP}      { return STOP_OP; }
{ADD}       { return ADD_OP; }
{SUB}       { return SUB_OP; }
{MULT}      { return MULT_OP; }
{TERM}      { return TERM_OP; }
{SUB_NUM}   { return SUB_NUM_OP; }
{DIV}       { return DIV_OP; }
{READ}      { return READ_OP; }
{PRINT}     { return PRINT_OP; }
{LIR}       { return LIR_OP; }
```

```
{IIR}        { return IIR_OP; }
{LOOP}       { return LOOP_OP; }
{STORE}      { return STORE_OP; }
{LOAD}       { return LOAD_OP; }
{TRANS}      { return TRANS_OP; }
{TRIM}       { return TRIM_OP; }

/* Identifier or label */
[_a-zA-Z][_a-zA-Z0-9]*  { yylval.sval = strdup(yytext); return IDENTIFIER;
}

/* Decimal number */
[0-9]+ { yylval.ival = atoi(yytext); return NUMBER; }

/* Single character */
. { return yytext[0]; }

%%

int yywrap(void) {
    return 1;
}
```

Assembler Data Structures

Assignment 4

For the assembly language described in the previous assignment, write a program that reads a file containing an assembly language program, and for each statement in the assembly program it prints the numeric opcode on the terminal in readable format, and in an output file in binary format. Also it should build the symbol table that contains each symbol encountered in the assembly program along with the line number in which that symbol is defined and the location counter value. After analyzing all the lines and printing the opcodes, the symbol table should be printed on the terminal. [You will have to define a mnemonic-table for this program where the number of machine language bytes to be produced for different assembly language statements will be mentioned. At runtime location counter should be initialized and updated appropriately.]

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


// Define a mnemonic-table for this program where the number
                      of machine language bytes to be produced
                      for different assembly language
                      statements will be mentioned.
static struct {
    char *mnemonic;
    int bytes;
} mnemonic_table[] = {
    {"STOP", 0},
    {"ADD", 1},
    {"SUB", 1},
    {"MULT", 1},
    {"LOAD", 1},
    {"STORE", 1},
    {"TRANS", 2},
    {"TRIM", 2},
    {"DIV", 1},
    {"READ", 1},
    {"PRINT", 1},
    {"LIR", 2},
    {"IIR", 2},
    {"LOOP", 2},
    {"START", 2},
    {"END", 0},
    {"LTORG", 0},
    {"DS", 2},
    {"DC", 1},
};

// Initialize the location counter.
static int location_counter = 1000;

// Open the input file.
FILE *input_file = fopen(argv[1], "r");

// Create an empty symbol table.
static struct {
    char *symbol;
    int address;
} symbol_table[1000];

// Read each line from the input file.
while (fgets(line, sizeof(line), input_file)) {
```

```
// Split the line into words.
char *words[10];
int word_count = 0;
char *token = strtok(line, " ");
while (token != NULL) {
    words[word_count++] = token;
    token = strtok(NULL, " ");
}

// If the first word is a mnemonic, then...
if (words[0] != NULL &&
                  mnemonic_table[word_count].mnemonic !=
                  NULL) {

    // Get the opcode for the mnemonic.
    int opcode = mnemonic_table[word_count].bytes;

    // Print the opcode on the terminal in readable
                  format.
    printf("Opcode: %s (%d)\n",
                  mnemonic_table[word_count].mnemonic,
                  opcode);

    // Write the opcode to the output file in binary
                  format.
    for (int i = 0; i < opcode; i++) {
        fputc(opcode, output_file);
    }

    // Update the location counter by the number of
                  machine language bytes for the opcode.
    location_counter += opcode;

} else if (words[0] != NULL && words[0][0] == 'L' &&
                  isalpha(words[0][1])) {

    // Add the symbol to the symbol table.
```

```
        symbol_table[word_count].symbol = words[0];
        symbol_table[word_count].address = location_counter;

    }
}

// Close the input file.
fclose(input_file);

// Print the symbol table on the terminal.
printf("Symbol Table:\n");
for (int i = 0; i < word_count; i++) {
    printf("%s : %d\n", symbol_table[i].symbol,
                         symbol_table[i].address);
}
```

Lexical Analyser

Assignment 3

Write a lexical analyser for the assembly language described in the file attached. The program should take as input a text file containing an assembly program and print the stream of token values corresponding to the items in the input file. For identifiers and numbers the actual input item should also appear within brackets along with the token value. e.g. the output may look like:

17 20(100) 21(AGAIN) 14 20(3) 21(TERM) ....

Write the program once in plain C, and once using flex utility (later).

[Create the file token.h to put the token definitions. In the plain C implementation, you should also define the mnemonic table containing the mnemonic strings and the corresponding numeric token values.]

Token.h

```
#ifndef TOKEN_H
#define TOKEN_H

enum Token {
    /* Mnemonic tokens */
    STOP = 0,
    ADD = 1,
    SUB = 2,
    MULT = 3,
    LOAD = 4,
    STORE = 5,
    TRANS = 6,
    TRIM = 7,
    DIV = 8,
    READ = 9,
    PRINT = 10,
    LIR = 11,
    IIR = 12,
    LOOP = 13,

    /* Directive tokens */
    START = 14,
    END = 15,
    LTORG = 16,

    /* Other tokens */
    IDENT = 17,
    NUMBER = 18,
    END_OF_FILE = 19
};

/* Mnemonic table */
struct Mnemonic {
    char *mnemonic;
    enum Token token;
};

/* Define the mnemonic table */
static struct Mnemonic mnemonics[] = {
    {"STOP", STOP},
    {"ADD", ADD},
    {"SUB", SUB},
    {"MULT", MULT},
    {"LOAD", LOAD},
    {"STORE", STORE},
    {"TRANS", TRANS},
    {"TRIM", TRIM},
    {"DIV", DIV},
    {"READ", READ},
    {"PRINT", PRINT},
    {"LIR", LIR},
    {"IIR", IIR},
    {"LOOP", LOOP},
    {"START", START},
    {"END", END},
    {"LTORG", LTORG},
    {NULL, 0}
};

#endif
```

Lexer.c

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

// Mnemonic table
struct Mnemonic {
    char name[10];
    int code;
} mnemonic_table[] = {
    {"STOP",  0},
    {"ADD",   1},
    {"SUB",   2},
    {"MULT",  3},
    {"LOAD",  4},
    {"STORE", 5},
    {"TRANS", 6},
    {"TRIM",  7},
    {"DIV",   8},
    {"READ",  9},
    {"PRINT", 10},
    {"LIR",   11},
    {"IIR",   12},
    {"LOOP",  13},
    {"START", 14},
    {"END",   15},
    {"LTORG", 16},
    {"G",     17},
    {"ONE",   18},
    {NULL,    -1}
};

// Function to check if a character is a valid identifier character
int is_identifier_char(char c) {
    return (c >= 'A' && c <= 'Z') || (c >= '0' && c <= '9') || c == '_';
}

// Function to check if a string is a valid identifier
int is_identifier(char *str) {
    if (*str == '\0') return 0;  // Empty string is not a valid identifier
    while (*str) {
        if (!is_identifier_char(*str)) return 0;
        str++;
    }
    return 1;
}

// Function to check if a string is a valid number
int is_number(char *str) {
    if (*str == '\0') return 0;  // Empty string is not a valid number
    while (*str) {
        if (*str < '0' || *str > '9') return 0;
        str++;
    }
    return 1;
}

// Function to lookup a mnemonic in the mnemonic table and return its
token code
int lookup_mnemonic(char *mnemonic) {
    int i = 0;
    while (mnemonic_table[i].name != NULL) {
        if (strcmp(mnemonic_table[i].name, mnemonic) == 0) {
            return mnemonic_table[i].code;
        }
        i++;
    }
    return -1;  // Mnemonic not found in table
}

// Function to get the next token from the input stream
int get_token(FILE *input, char *token, char *value) {
    int c;
    do {
        c = fgetc(input);
    } while (c == ' ' || c == '\t' || c == '\n');

    // Check for end of file
    if (c == EOF) {
        return TOKEN_EOF;
    }

    // Check for identifier or mnemonic
    if (is_identifier_char(c)) {
        int i = 0;
        while (is_identifier_char(c)) {
            token[i++] = c;
            c = fgetc(input);
        }
        token[i] = '\0';
        ungetc(c, input);
        int code = lookup_mnemonic(token);
        if (code == -1) {
            // Identifier
            strcpy(value, token);
            return TOKEN_IDENTIFIER;
        } else {
            // Mnemonic
            return code;
        }
    }

    // Check for number
    if (c >= '0' && c <= '9') {
        int i = 0;
        while (c >= '0' && c <= '9') {
           int lex(FILE* file) {
    int token_count = 0;
    char c, id[MAX_ID_LEN], num[MAX_NUM_LEN];
    while ((c = fgetc(file)) != EOF) {
        if (isspace(c)) {
            continue;
        } else if (c == ':') {
            tokens[token_count].type = COLON;
            token_count++;
        } else if (c == ',') {
            tokens[token_count].type = COMMA;
            token_count++;
        } else if (c >= '0' && c <= '9') {
            int i = 0;
            while (c >= '0' && c <= '9') {
                num[i++] = c;
                c = fgetc(file);
            }
            num[i] = '\0';
            ungetc(c, file);
            tokens[token_count].type = NUM;
            tokens[token_count].value.num_value = atoi(num);
            token_count++;
        } else if (isalpha(c)) {
            int i = 0;
            while (isalnum(c) && i < MAX_ID_LEN - 1) {
                id[i++] = c;
                c = fgetc(file);
            }
            id[i] = '\0';
            ungetc(c, file);
            if (strcmp(id, "STOP") == 0) {
                tokens[token_count].type = STOP;
            } else if (strcmp(id, "ADD") == 0) {
                tokens[token_count].type = ADD;
            } else if (strcmp(id, "SUB") == 0) {
                tokens[token_count].type = SUB;
            } else if (strcmp(id, "MULT") == 0) {
                tokens[token_count].type = MULT;
            } else if (strcmp(id, "LOAD") == 0) {
                tokens[token_count].type = LOAD;
            } else if (strcmp(id, "STORE") == 0) {
                tokens[token_count].type = STORE;
            } else if (strcmp(id, "TRANS") == 0) {
                tokens[token_count].type = TRANS;
            } else if (strcmp(id, "TRIM") == 0) {
                tokens[token_count].type = TRIM;
            } else if (strcmp(id, "DIV") == 0) {
                tokens[token_count].type = DIV;
            } else if (strcmp(id, "READ") == 0) {
                tokens[token_count].type = READ;
            } else if (strcmp(id, "PRINT") == 0) {
                tokens[token_count].type = PRINT;
            } else if (strcmp(id, "LIR") == 0) {
                tokens[token_count].type = LIR;
            } else if (strcmp(id, "IIR") == 0) {
                tokens[token_count].type = IIR;
            } else if (strcmp(id, "LOOP") == 0) {
                tokens[token_count].type = LOOP;
            } else if (strcmp(id, "START") == 0) {
                tokens[token_count].type = START;
            } else if (strcmp(id, "END") == 0) {
                tokens[token_count].type = END;
            } else if (strcmp(id, "LTORG") == 0) {
                tokens[token_count].type = LTORG;
            } else {
                tokens[token_count].type = ID;
                strcpy(tokens[token_count].value.id_value, id);
            }
            token_count++;
        }
    }
    tokens[token_count].type = END_OF_FILE;
    return token_count;
}
}
```

Assignment 2: Mnemonic recognition

Write a program that reads an assembly language program (assume any assembly language) and for each mnemonic found prints a corresponding numeric opcode on the terminal. [Define numeric opcodes for a set of mnemonics]

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

#define MAX_MNEMONIC_LENGTH 10

struct mnemonic {
  char *mnemonic;
  int opcode;
};

struct mnemonic mnemonics[] = {
  { "ADD", 0x01 },
  { "SUB", 0x02 },
  { "MUL", 0x03 },
  { "DIV", 0x04 },
  { "MOV", 0x05 },
  { "CMP", 0x06 },
  { "JMP", 0x07 },
  { "JZ", 0x08 },
  { "JNZ", 0x09 },
  { "HLT", 0x0A },
};

int main() {
  char line[1000];
  int num_mnemonics = sizeof(mnemonics) / sizeof(mnemonics[0]);

  while (fgets(line, 1000, stdin) != NULL) {
    char *token;
    int opcode = -1;

    token = strtok(line, " ");
    if (token == NULL) {
      continue;
    }
    for (int i = 0; i < num_mnemonics; i++) {
      if (strcmp(token, mnemonics[i].mnemonic) == 0) {
        opcode = mnemonics[i].opcode;
        break;
      }
    }

    if (opcode != -1) {
      printf("%d\n", opcode);
    }
  }

  return 0;
}
```

Text and Binary File Handling

Assignment 1

This assignment is meant as an exercise to strengthen your skills of handling text and data files using C language.

Write a C program that reads text from a file and prints on the terminal each input line, preceded by the line number. The output will look like -
1 This is the first trial line in the file,
2 and this is the second line.
Try the problem once using fgetc() function and once using fgets() function for reading the input. Why is fread() not suitable for this purpose?
Do not ignore the value returned by the functions fgetc() and fgets(). After this exercise you should be comfortable with the formatted input and output functions of C.

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

int main() {
  FILE *file = fopen("input.txt", "r");
  if (file == NULL) {
    printf("Error opening file\n");
    return 1;
  }

  int line_number = 1;
  char line[1000];
  while (fgets(line, 1000, file) != NULL) {
    printf("%d %s\n", line_number, line);
    line_number++;
  }

  fclose(file);

  return 0;
}
```

Write a program that takes from the user the name of a file and a "field-number", and then reads that file and for each line in that file prints on the terminal word at position "field-number". For example if there are the following lines in the specified file -
C is a programming language.
lex produces a lexical analyser
cc is a compiler
and if the field-number specified is 4, then the output of the program is -
programming
lexical
(NULL)
compiler

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

int main() {
  char *filename;
  int field_number;

  printf("Enter the name of the file: ");
  fgets(filename, 1000, stdin);
  filename[strlen(filename) - 1] = '\0';

  printf("Enter the field number: ");
  scanf("%d", &field_number);

  FILE *file = fopen(filename, "r");
  if (file == NULL) {
    printf("Error opening file\n");
    return 1;
  }

  char line[1000];
  while (fgets(line, 1000, file) != NULL) {
    char *token;
    token = strtok(line, " ");
    for (int i = 0; i < field_number - 1; i++) {
      token = strtok(NULL, " ");
    }

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

  fclose(file);

  return 0;
}
```

Write a C program that reads the names (as character strings of length upto 20 bytes) and corresponding roll-numbers (as integers) of 10 students from the user and stores

them in a file whose name is specified by the user-
o in text format

o in binary format

After running the program check the size of the file created using ls -l command. Also see the content of the binary file using the command
od -c filename.
After this exercise you should be clear about the difference between a text file and a binary file.

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

int main() {
  char *filename;
  int i;

  printf("Enter the name of the file: ");
  fgets(filename, 1000, stdin);
  filename[strlen(filename) - 1] = '\0';

  FILE *file = fopen(filename, "w");
```

```
 if (file == NULL) {
   printf("Error opening file\n");
   return 1;
 }

 for (i = 0; i < 10; i++) {
   char name[20];
   int roll_number;

   printf("Enter the name of student %d: ", i + 1);
   fgets(name, 20, stdin);
   name[strlen(name) - 1] = '\0';

   printf("Enter the roll number of student %d: ", i + 1);
   scanf("%d", &roll_number);

   fprintf(file, "%s %d\n", name, roll_number);
 }

 fclose(file);

 return 0;
}
```

Write a C program that works like the od program of LINUX (UNIX).

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

int main(int argc, char *argv[]) {
 char *filename;
 int format;
 int start;
 int end;

 if (argc < 2) {
   printf("Usage: od [options] filename\n");
   return 1;
 }

 filename = argv[1];

 for (int i = 2; i < argc; i++) {
   if (argv[i][0] == '-') {
     format = argv[i][1];
     break;
   }
 }

 if (format == '\0') {
   format = 'o';
 }

 if (argc > 3) {
   start = atoi(argv[2]);
   end = atoi(argv[3]);
 } else {
   start = 0;
   end = -1;
 }

 FILE *file = fopen(filename, "rb");
 if (file == NULL) {
   printf("Error opening file\n");
   return 1;
 }

 char buffer[1024];
 int bytes_read;

 while ((bytes_read = fread(buffer, sizeof(char), 1024, file)) != 0) {
   for (int i = 0; i < bytes_read; i++) {
     switch (format) {
       case 'o':
         printf("%03o ", buffer[i]);
         break;
       case 'x':
         printf("%02x ", buffer[i]);
         break;
       case 'd':
         printf("%d ", buffer[i]);
         break;
       case 'c':
         printf("%c ", buffer[i]);
         break;
       default:
         printf("Unknown format\n");
         return 1;
      }
    }

    printf("\n");
  }

  fclose(file);

  return 0;
}
```

Write a C program that works like the strings program of LINUX (UNIX).

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

int main(int argc, char *argv[]) {
  char *filename;
  char *buffer;
  size_t buffer_size;

  if (argc < 2) {
    printf("Usage: strings [options] filename\n");
    return 1;
  }

  filename = argv[1];

  FILE *file = fopen(filename, "rb");
  if (file == NULL) {
    printf("Error opening file\n");
    return 1;
  }

  fseek(file, 0, SEEK_END);
  long file_size = ftell(file);
  fseek(file, 0, SEEK_SET);

  buffer_size = file_size * 2;
  buffer = (char *)malloc(buffer_size);
  if (buffer == NULL) {
    printf("Error allocating buffer\n");
    return 1;
  }
  size_t bytes_read = fread(buffer, sizeof(char), buffer_size, file);
  if (bytes_read != buffer_size) {
    printf("Error reading file\n");
    return 1;
  }

  char *current = buffer;
  while (*current != '\0') {
    if (isprint(*current)) {
      printf("%s\n", current);
      while (*current != '\0' && isprint(*current)) {
        current++;
      }
    }
    current++;
  }

  free(buffer);
  fclose(file);

  return 0;
}
```